MASTER EN SCIENCES INFORMATIQUES

La Transformée en ondelettes: de la théorie à la compression d'images

Reinbold, Pierre

Award date:
2000

La Transformée en Ondelettes : de la théorie à la compression d'images.

Reinbold Pierre
1999-2000

La Transformée en Ondelettes : de la théorie à la compression d'images

Travail réalisé sous la direction de Mr J. Fichefet et Mme S. Saadaoui en vue de l'obtention du grade de Licencié en Informatique

Reinbold Pierre

Année académique 1999-2000

Abstract

La Transformée en Ondelettes est devenue en quelques années un sujet de recherche très débattu. On ne compte plus aujourd'hui les applications qui utilisent cette technique. Issue de la théorie de Fourier, la Transformation en Ondelettes se caractérise par le développement des fonctions de $\mathbf{L}^2(\mathbb{R})$ sur une base d'ondelettes. Ces ondelettes sont elles-mêmes des dilatations et des translations d'une fonction unique que l'on nomme l'Ondelette Mère.

Dans ce travail, je fourni une présentation des principaux concepts sous-jacents à la théorie de la Transformée en Ondelettes. Je présente ensuite la Transformée en Ondelettes Discrète ou Représentation en Ondelettes Orthogonales qui fut introduite il y a une dizaine d'années. Il s'agit d'un algorithme permettant de calculer une représentation d'un signal en bandes de fréquences indépendantes. Cette représentation est particulièrement utile pour le traitement d'images. Dans ce travail, je montre un programme qui utilise cette technique afin de compresser des images naturelles. Ce programme fait partie d'une toolkit écrite en JAVA et en FORTRAN90 pour servir au programmeur.

Enfin, je présente les algorithmes actuels de compression d'image par Transformée en Ondelettes et leurs performances comparées par rapport au standard JPEG.

♦

The Wavelet Transform became in a few years a very discussed subject of research. One does not count today any more the applications which use this technique. Resulting from the theory of Fourier, the Wavelet Transform is characterized by the development of the function of $\mathbf{L}^2(\mathbb{R})$ *on a wavelets basis. These wavelets are themselves of dilations and translations of a single function which one names the mother wavelet.*

In this work, I provided a presentation of the main concepts of the theory of the Wavelet Transform. I present then the Discrete Wavelet Transform or Orthogonal Wavelet Representation which was introduced ten years ago. It is about an algorithm making it possible to calculate a representation of a signal in independent frequency bands. This representation is particularly useful for image processing. In this work, I show a program which uses this technique in order to compress natural images. This program forms part of a toolkit written in JAVA and FORTRAN90 to be used for the programmer.

Lastly, I present the current algorithms of image compression per Wavelet Transform and their performances compared to standard JPEG compression.

Remerciements

Si la réalisation d'un mémoire est avant tout une aventure personnelle, elle n'en est pas solitaire pour autant. Ce travail n'aurait pas été possible sans le concours de plusieurs personnes, ce sont celles-ci que je tiens à remercier ici. Tout d'abord, je voudrais exprimer ma gratitude à Mme Saadaoui, qui m'a accompagné dans cette réalisation, pour sa gentillesse et ses conseils et à Mr Fichefet qui me parrainait. Je n'oublierai pas non plus la sympathie et la disponibilité du personnel de la bibliothèque des sciences exactes de Louvain-La-Neuve. Que tous les passionnés des Ondelettes *on the web* soient remerciés pour les fructueux échanges que nous avons eus. Je remercie aussi Vincent Letocart pour son aide *logistique*. Je voudrais exprimer également ma gratitude à ma famille et mes amis (hello le **Yiti**$^{++}$) pour leur humour, leur patience et leur soutien. Enfin, je remercie Julie sans qui ce travail n'aurait tout simplement jamais pu exister.

Table des matières

Introduction

Lorsqu'en 1807, Joseph Fourier proposa de décomposer les fonctions périodiques en sommes finies d'exponentielles complexes, il ignorait certainement toute l'importance que son geste allait prendre dans les années à venir. En effet, il n'est pas besoin de rappeller l'importance énorme qu'a aujourd'hui la Transformée de Fourier dans une variété inimaginable de domaines.

Cependant, dans un coin du nouvel univers que Fourier venait d'ouvrir se tapissaient les Ondelettes. Il fallut une centaine d'années pour qu'un pionnier les mentionne : il s'agit de A. Haar dans sa thèse de doctorat en 1909. Les Ondelettes ne semblent pas avoir intéressé énormément les chercheurs jusque dans les années 80. Nous devons à Stéphane Mallat, Yves Meyer et Ingrid Daubechies d'avoir relancé l'intérêt vers 1985 en étoffant grandement la théorie et les applications possibles de celles-ci. Depuis leurs travaux, le sujet a véritablement pris une ampleur considérable et s'introduit dans un grand nombre de domaines. Citons par exemple le traitement de signaux bruités, l'intelligence artificielle, la synthèse artificielle des sons, l'analyse d'électrocardiogrammes [10], la détection et la prédiction de tremblements de terre [7] et bien sûr, le traitement d'images. Cette dernière activité, qui fait aussi l'objet de ce mémoire, est sans doute celle où les Ondelettes ont montré le plus leurs performances. Aujourd'hui, on s'en sert pour stocker les images d'empreintes digitales au FBI, transmettre et stocker les images satellites, et notamment de Meteosat [22, 12], analyser des images médicales [1] aussi bien issues de tomographes [11] que de scanner RMN [47]. On s'en sert également pour compresser les images naturelles, ce dont je vais parler dans la suite de ce travail. Le lecteur pourra trouver une description plus complète des applications de la Transformée en Ondelettes dans l'annexe C.

Lorsque Madame Saadaoui m'a proposé de travailler sur les Ondelettes, nous ne savions ni l'un ni l'autre de quoi il retournait exactement mais le défi m'a paru intéressant. Aucun autre travail n'avait été fait à l'Institut sur cette matière et nous envisagions celui-ci comme un premier pas susceptible d'en précéder bien d'autres tant les applications semblaient (et sont) nombreuses.

Malheureusement, nous nous sommes rapidement heurtés aux difficultés liées à la relative jeunesse de cette théorie. La documentation était très pénible à trouver dans un premier temps. De plus, il n'y a pratiquement pas d'introduction à la théorie faite pour quelqu'un ne disposant pas de connaissances mathématiques très poussées.

Nos objectifs de départ étaient donc de faire un travail comportant deux par-

ties : une partie théorique, sur la Transformée de Fourier elle-même et une partie pratique qui en montre une application. Le défrichage de la théorie élégante mais touffue de la Transformée en Ondelettes me prit beaucoup plus de temps que nous ne pouvions l'imaginer au départ. Ce qui retarda d'autant le travail sur l'application. C'est donc tardivement que j'ai découvert la Transformée en Ondelettes Discrète et toute la théorie qui lui est sous-jacente. Le tout était également à assimiler puisque toutes les applications se font sur base de cet algorithme.

Ainsi, à ce moment, j'ai pu programmer une toolkit JAVA permettant de constater l'efficacité étonnante des Ondelettes pour la compression d'images. J'ai également programmé les mêmes fonctionnalités en FORTRAN90 pour avoir plus d'efficacité au niveau du temps d'exécution. Je n'ai cependant pas programmé les dernières étapes de compression effectives des images avec la Transformée en Ondelettes Discrète. Celles-ci sont constituées d'algorithmes tout à fait originaux et très variés qui sont autant de matières indépendantes à assimiler. Je me contenterai donc de présenter le premier et le plus important d'entre eux et de mentionner les résultats que l'on peut obtenir avec les autres.

Ce mémoire se découpe donc en trois parties principales. La première sera consacrée à la Transformée en Ondelettes Continue. J'introduirai celle-ci en passant par une courte révision de notions de traitement du signal et des outils usuels tels que la Transformée de Fourier et la Transformée de Fourier Fenêtrée. Je présenterai ensuite les résultats fondamentaux de la théorie afin que le lecteur puisse avoir un bon aperçu des concepts manipulés.

J'exposerai ensuite largement la théorie de la Transformée en Ondelettes Discrète telle qu'elle a été présentée par Stéphane Mallat. Ceci constituera la seconde partie de ce travail.

Je présenterai alors la compression d'images telle qu'on la pratique avec la Transformée en Ondelettes Discrète pour ensuite montrer les programmes que j'ai réalisés avec leurs résultats. L'algorithme fondamental pour les dernières étapes de compression sera présenté avec l'état de l'art en terme de performances pour les codeurs actuels.

Chapitre 1

Transformée en Ondelettes Continue

Comme prévu, ce chapitre va constituer une présentation de la théorie de la Transformée en Ondelettes. Mon propos n'est pas d'être exhaustif : une matière aussi foisonnante que celle des Ondelettes remplirait des volumes entiers. Plus simplement, j'ai abordé la théorie dans l'esprit d'une exposition des concepts fondamentaux, adressée au lecteur non mathématicien. Mon but premier est d'être utile et rapidement compréhensible. Le lecteur intéressé trouvera toujours des renvois aux références contenant tous les détails voulus.

J'ai donc considéré que le lecteur ne possède que des notions de base en mathématiques et très peu en traitement du signal. C'est pourquoi je commencerais par un rapide survol des concepts fondamentaux de cette dernière matière. On trouvera dans l'annexe A toutes les notations que j'ai utilisé ainsi qu'une synthèse des concepts essentiels à la bonne compréhension du texte.

L'introduction de la Transformée en Ondelettes se fera via une présentation de la Transformée de Fourier et de la Transformée de Fourier Fenêtrée qui permettent de mettre en valeur les avantages importants de l'utilisation des Ondelettes.

Enfin, la théorie de la Transformée en Ondelettes Continue proprement dite sera exposée dans l'esprit que j'ai décrit plus haut.

La structure de ce chapitre reprend les grandes lignes du tutorial de Robi Polikar [32]. Les développements mathématiques sont principalement tirés des chapitres 1 et 2 de l'excellent livre d'Ingrid Daubechies [6] et d'autres présentations générales [21, 15, 48, 8]. Les illustrations proviennent également de ces documents.

1.1 Notions de traitement du signal : Transformée de Fourier et Transformée de Fourier Fenêtrée

La plupart des signaux que nous rencontrons sont des signaux dépendant du temps. Ainsi, la différence de potentiel aux bornes d'une prise de courant se présente comme une sinusoïde que l'on caractérisera par sa fréquence : 50 Hz (partout dans le monde sauf aux USA). En général, les signaux ne sont pas aussi simples.

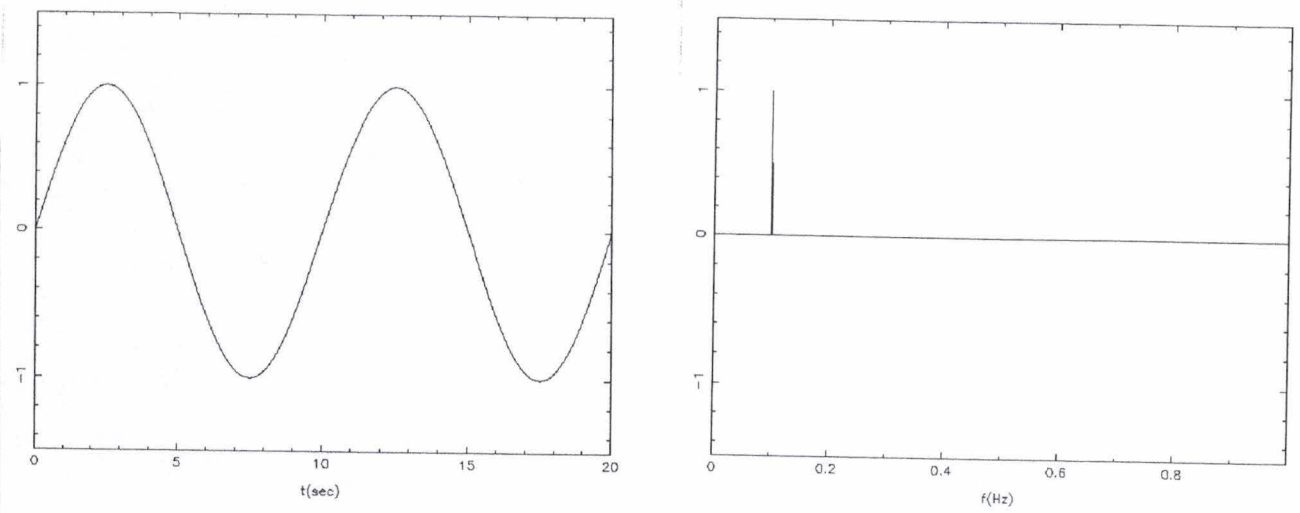


FIG. 1.1 – *Une sinusoïde et son spectre*

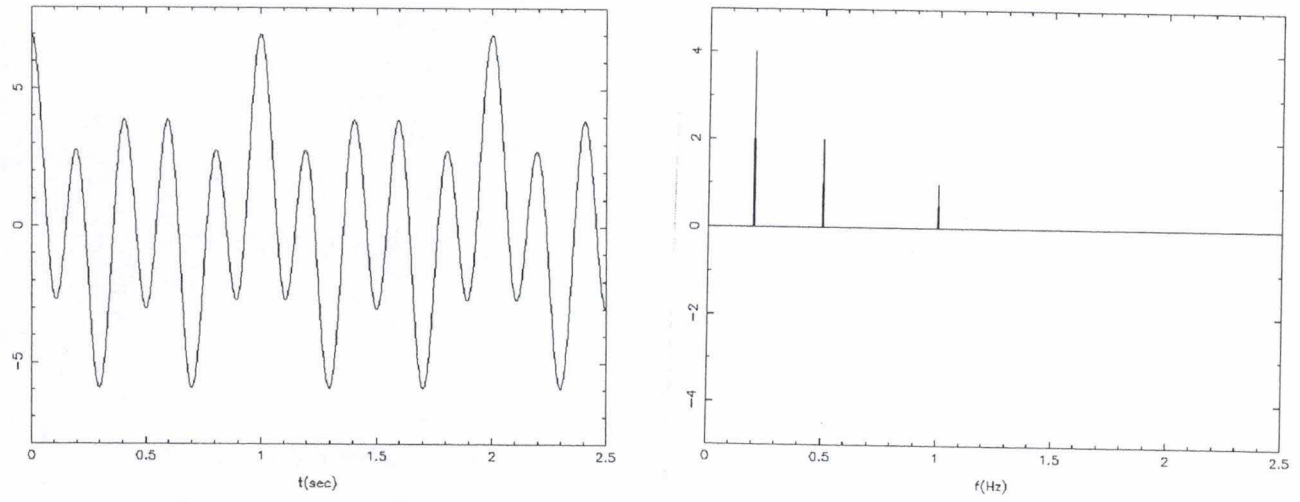


FIG. 1.2 – *Signal contenant 3 fréquences et son spectre*

Ils contiennent la superposition de plusieurs fréquences qui ont plus ou moins d'importance. C'est pourquoi, lorsqu'on analyse un signal brut, l'une des informations capitales que l'on peut tenter d'obtenir est son contenu en fréquences : son spectre. Ainsi, si je prends le signal de la figure 1.1 , le spectre correspondant est très simple et comprend un pic à la fréquence de la sinusoïde. Cependant, dans le cas de la figure 1.2 , il est plus difficile de constater que ce signal est composé de trois fréquences, comme indiqué sur le spectre en-dessous. Cela signifie que ce signal est composé de la somme de trois sinusoïdes dont les amplitudes relatives sont représentées par les amplitudes des pics correspondants en fréquence. Ainsi l'expression mathématique de ce signal est

$$f(t) = \cos 2\pi t + 2\cos(2\pi.2.t) + 4\cos(2\pi.5.t)$$

Il existe une opération mathématique très répandue et largement étudiée qui permet d'obtenir le contenu fréquentiel d'un signal et que l'on nomme Transformée de Fourier. C'est au XIXème siècle que le mathématicien français Joseph Fourier montra qu'un signal périodique est exprimable comme une somme infinie d'exponentielles complexes correctement pondérées. Chaque terme de la somme correspondant à une fréquence dont le coefficient donne l'importance dans le spectre. Cette idée a été généralisée pour les signaux non périodiques et puis pour les signaux discrets (périodiques ou non). Aujourd'hui, un algorithme très rapide, nommé Transformée de Fourier Rapide, permet d'obtenir la Transformée de Fourier d'un signal de manière très efficace.

L'étude des signaux discrets et de durée limitée dans le temps (c'est à dire tous les signaux réels, échantillonnés sur un temps fini) ne se fait pas sans quelques artifices mathématiques. Cependant, dans le cadre de cette approche introductive, nous allons nous restreindre au cas de fonctions continues et définies de $-\infty$ à $+\infty$. Ceci sera suffisant pour illustrer les propriétés saillantes de la Transformée de Fourier.

Ainsi, la Transformée de Fourier d'un signal $x(t)$ se définit par

$$(\mathcal{F}x)(f) = \int_{-\infty}^{+\infty} x(t)\ e^{-2i\pi ft}\ \mathrm{d}t = \hat{x}(f)$$

Et, à partir de la Transformée de Fourier d'un signal, on peut reconstruire celui-ci en lui appliquant une transformée inverse :

$$(\mathcal{F}^{-1}\hat{x})(t) = \int_{-\infty}^{+\infty} \hat{x}(f)\ e^{2i\pi ft}\ \mathrm{d}f = x(t)$$

La composante f du spectre est donc obtenue en multipliant le signal par $e^{-2i\pi ft}$ et en intégrant ceci sur tous les temps de $-\infty$ à $+\infty$. Par conséquent, la Transformée de Fourier ne rend pas compte de la manière dont la composition en fréquences du signal a évolué au cours du temps. Je m'explique : si un signal présente depuis un temps t_1 à un temps t_2 une composante de fréquence f et d'amplitude A, le spectre de ce signal obtenu par Transformée de Fourier va rendre compte de cette fréquence mais pas du moment où elle est apparue. Ainsi, un autre signal présentant la même composante de fréquence à la même amplitude et pendant un temps identique mais à partir de $t_3 \neq t_1$ aura le même spectre de Fourier.

Donc, la Transformée de Fourier nous renseigne si une composante fréquentielle est présente ou pas indépendamment du moment où elle a été effectivement présente dans le signal. Ceci n'a aucune importance pour les signaux stationnaires mais on perd effectivement de l'information pour les signaux qui ne le sont pas.

En guise d'illustration, observons les deux signaux de la figure 1.3 tirés de [32]. Le premier présente quatre fréquences à tout moment, ce dont son spectre rend compte. Le second présente les mêmes fréquences mais à des moments différents. On constate que, même si le spectre est fortement bruité (à cause des changements de fréquence

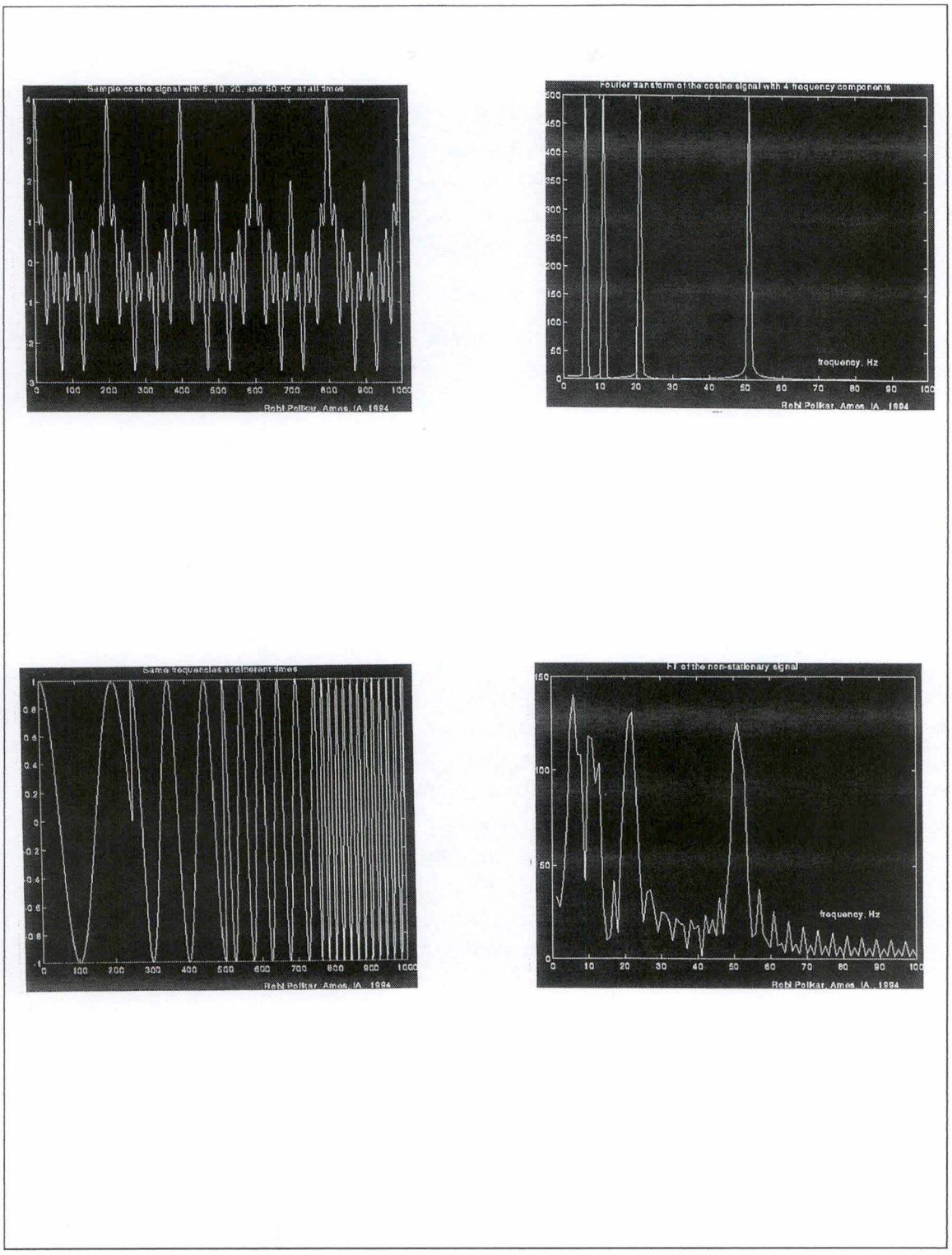


FIG. 1.3 – *Deux signaux présentant les mêmes fréquences mais à des moments différents. On constate sur leurs Transformées de Fourier que les spectres sont similaires.*

brutaux)[1], les quatre composantes principales sont présentes de la même manière que pour le premier signal.

Il faut donc trouver une autre solution si l'on veut étudier l'évolution du contenu fréquentiel d'un signal. Une solution possible est celle de la Transformée de Fourier Fenêtrée. Cette transformation se propose de découper le signal en "tranches" pendant lesquelles celui-ci pourra être considéré comme stationnaire et de prendre la Transformée de Fourier dans chacune des tranches. On exprime ceci par la formule suivante :

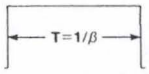
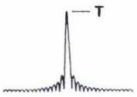
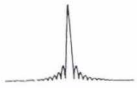
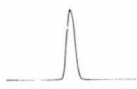
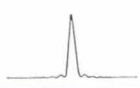
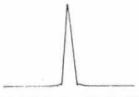
$$(\mathcal{F}^{win}x)(f,t') = \int x(t)win(t-t')\, e^{-2i\pi ft}\ \mathrm{d}t$$

Cette fois, la transformée est à deux dimensions : temps et fréquence, ce qui laisse présager l'obtention d'informations sur ces deux grandeurs. En fait, il s'agit simplement de la Transformée de Fourier de $x(t)win(t-t')$. $win(t)$ est une fonction de fenêtrage que l'on considèrera réelle (elle peut être complexe mais on doit alors utiliser le complexe conjugé dans la formule). Ce type de fonctions est très largement répandu en traitement du signal et celles-ci ont fait l'objet d'études poussées. Dans le cadre de ce mémoire, nous retiendrons simplement qu'il s'agit de fonctions qui permettent de sélectionner une partie d'un signal par multiplication avec celui-ci. La figure 1.4 montre quelques exemples de fenêtres couramment utilisées avec leurs Transformées de Fourier. Ainsi, la Transformée de Fourier Fenêtrée consiste à sélectionner une partie du signal centrée autour de t' avec une largeur égale à celle de la fenêtre et à prendre la Transformée de Fourier du résultat. On peut obtenir une représentation temps-fréquence du signal en parcourant tout l'axe des temps avec t'. Cette procédure est illustrée par la figure 1.5 .

La figure 1.6 donne un signal divisé en quatre intervalles de 250 msec dans lesquels on trouve des sinusoïdes de 300, 200, 100 et 50 Hz. La Transformée de Fourier Fenêtrée, en trois dimensions, nous donne la représentation temps-fréquence du signal (les unités sont arbitraires). On peut voir que cette figure est symétrique par rapport au centre de l'axe des fréquences, ce qui est une propriété normale de la Transformée de Fourier. Si on examine par exemple la partie droite, on constate la présence des quatre "pics" correspondant aux différentes fréquences du signal et situés à des temps différents, comme on le voulait.

Cependant, cet exemple bien choisi ne doit pas masquer le défaut principal de la Transformée de Fourier Fenêtrée ; la résolution temps-fréquence. Ce type de problème est bien connu en traitement du signal et est lié à une propriété fondamentale qui du temps et de la fréquence : il est impossible d'avoir simultanément une résolution infinie dans ces deux grandeurs. Je vais ici me borner à quelques considérations générales sur ce problème, le lecteur intéressé pourra se référer à la bibliographie. Ainsi, dans le cas de la Transformée de Fourier, la résolution fréquentielle est infinie

[1] Les changements abrupts dans le signal en temps introduisent du bruit et des hautes fréquences. Tout ceci pourrait être filtré par des techniques idoines

Unity Amplitude Window	Shape Equation	Frequency Domain Magnitude	Major Lobe Height	Highest Side Lobe (dB)	Band-width (3 dB)	Theoretical Roll-Off (dB/Octave)
Rectangle 	A=1 for t=0 to T		T	−13.2	0.86β	6
Extended Cosine Bell 	A=0.5(1 cos 2π5t/T) for t=0 to T/10 and t=9T/10 to T A=1 for t=T/10 to 9T/10		0.9 T	−13.5	0.95β	18 (beyond 5β)
Half Cycle Sine 	A=sin 2π0.5t/T for t=0 to T		0.64 T	−22.4	1.15β	12
Triangle 	A=2t/T for t=0 to T/2 A=2t/T−2 for t=T/2 to T		0.5 T	−26.7	1.27β	12
Cosine² (Hanning) 	A=0.5(1−cos 2πt/T) for t=0 to T		0.5 T	−31.6	1.39β	18
Half Cycle Sine³ 	A=sin³ 2π0.5t/T for t=0 to T		0.42 T	−39.5	1.61β	24
Hamming 	A=0.08+0.46(1−cos 2πt/T) for t=0 to T		0.54 T	−41.9	1.26β	6 (beyond 5β)
Cosine⁴ 	A=(0.5(1−cos 2πt/T)² for t=0 to T		0.36 T	−46.9	1.79β	30
Parzen 	A=1−6(2t/T−1)²+6|2t/T−1|³ for t=T/4 to 3T/4 A=2(1−|2t/T−1|)³ for t=0 to T/4 and t=3T/4 to T		0.37 T	−53.2	1.81β	24

C1754-79

FIG. 1.4 – *Quelques exemples de fenêtres utilisées en traitement du signal avec leurs principales caractéristiques.*

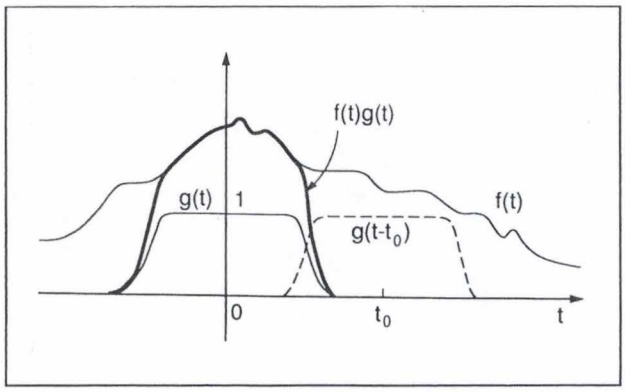


FIG. 1.5 – *La Transformée de Fourier Fenêtrée consiste à multiplier le signal et à prendre la Transformée de Fourier du produit. On répète ensuite l'opération en translatant la fenêtre.*

mais nous n'avons aucune information temporelle dans le domaine des fréquences. On sait que ces fréquences existent, on ne sait pas à quel moment. Cette propriété est liée au fait que l'intégration de la Transformée de Fourier va de $-\infty$ à $+\infty$. Dans le cas de la Transformée de Fourier Fenêtrée, nous réduisons l'intervalle d'intégration par un fenêtrage pour avoir des informations temporelles et donc, nous perdons une partie du signal. C'est ce qui rend la résolution en fréquence moins bonne : ce que l'on gagne d'un côté, on le perd de l'autre! Une fenêtre étroite voudra dire une résolution temporelle bonne mais une mauvaise résolution fréquentielle. Si on élargit la fenêtre, on gagne en résolution fréquentielle mais on perd en résolution temporelle. Ceci est illustré à la figure 1.7 tirée de [32].

Donc, le problème de fixer la taille de la fenêtre est à priori insoluble pour la Transformée de Fourier Fenêtrée et c'est ce qui en fait un outil peu utilisé.

1.2 La Transformée en Ondelettes Continue

1.2.1 Introduction : La Transformée en Ondelettes Continue et la Transformée de Fourier Fenêtrée

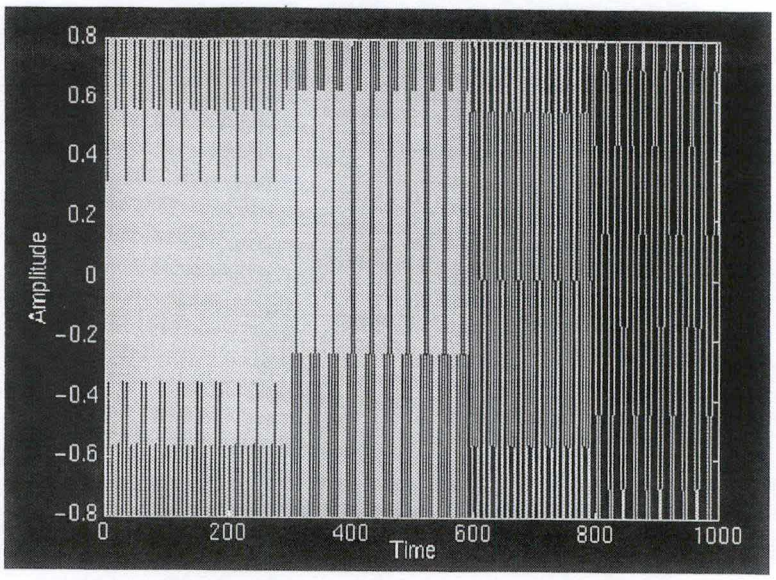
En guise d'introduction, anticipons un peu sur la théorie pour montrer les avantages de la Transformée en Ondelettes sur la Transformée de Fourier Fenêtrée.

La Transformée en Ondelettes de la fonction $x(t)$ se définit par la relation suivante :

$$(\mathcal{T}x)(a,b) = |a|^{-\frac{1}{2}} \int x(t)\, \psi\left(\tfrac{t-b}{a}\right) \, \mathrm{d}t$$

$$avec \int \psi(t)\, \mathrm{d}t = 0$$

Comme pour la Transformée de Fourier Fenêtrée, nous considérons que ψ est une



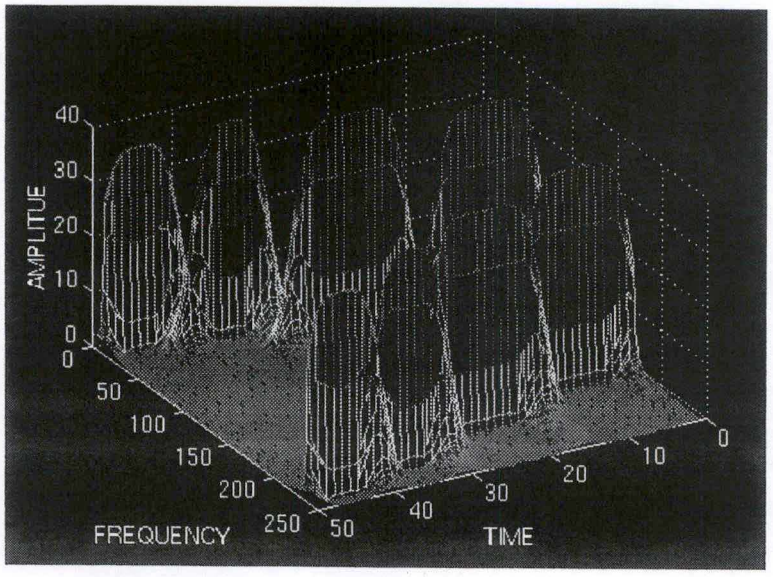


FIG. 1.6 – *Signal présentant quatre fréquences et sa Transformée de Fourier Fenêtrée.*

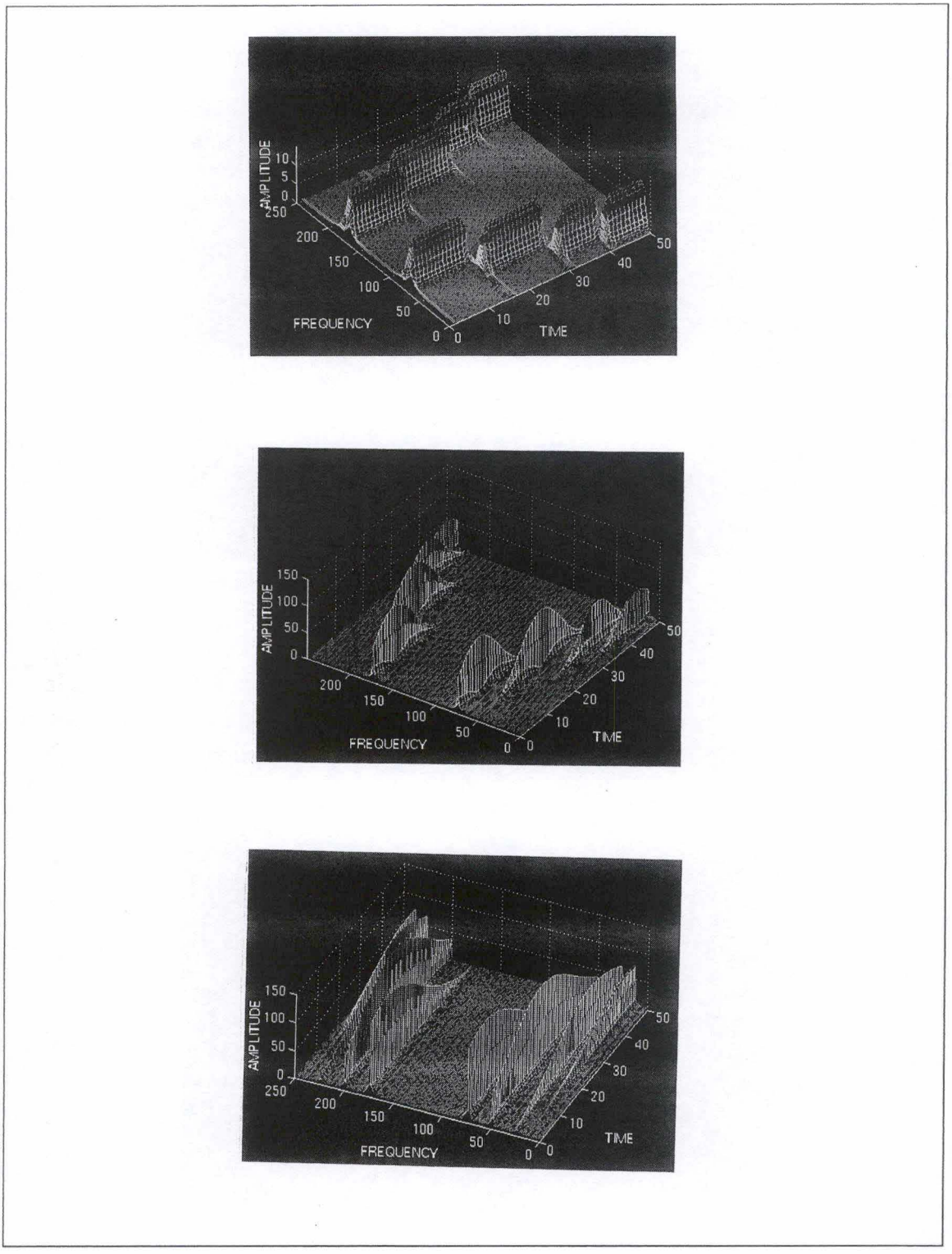


FIG. 1.7 – *Transformée de Fourier Fenêtrée du signal précédent avec trois largeurs de fenêtres différentes. Une première fenêtre, étroite, nous donne une bonne résolution en temps mais mauvaise en fréquence. Les deux spectres suivants, pris avec des fenêtres de plus en plus larges, montrent le gain de résolution en fréquence mais la perte en temps.*

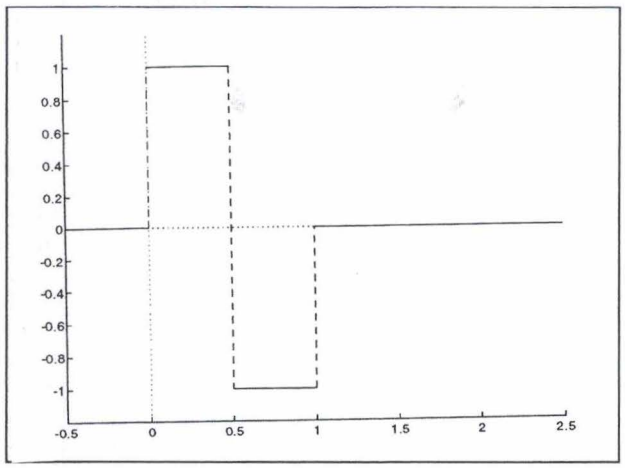


FIG. 1.8 – *Ondelette de Haar*

fonction réelle mais ψ peut être complexe, auquel cas, il faut utiliser le complexe conjugué dans l'intégrale de la Transformée en Ondelettes.

Dans cette formule, les fonctions $\psi^{a,b}(t) = |a|^{-\frac{1}{2}}\psi\left(\frac{t-b}{a}\right)$ sont appellées Ondelettes (Wavelets) et la Transformée en Ondelettes Continue consiste à développer $x(t)$ sur ces fonctions. Les Ondelettes consistent simplement en dilatations et translations d'une fonction unique $\psi(t)$ que l'on appelle l'Ondelette Mère (Mother Wavelet). Cette fonction doit répondre, pour des raisons qui apparaîtront dans la suite, à la condition $\int \psi(t)\mathrm{d}t = 0$, ce qui l'oblige à être "oscillante" d'une façon ou d'une autre. L'Ondelette la plus simple est l'Ondelette de Haar qui est présentée sur la figure 1.8 tirée de [48]. Mais il existe d'autres Ondelettes dont les propriétés sont très différentes et dont les applications varient. Citons encore le "Chapeau mexicain" (Mexican Hat) qui est simplement la dérivée seconde d'une gaussienne

$$\psi(t) = (1 - t^2)\ e^{-\frac{t^2}{2}}$$

La figure 1.9 donne son graphe. Cette fonction est bien localisée en temps et en fréquence. Ainsi, le paramètre a que l'on nomme le paramètre d'échelle (scale) va faire en sorte que $\psi^{a,o}(t)$ couvre des gammes de fréquences différentes. Si $|a|$ est faible, $\psi^{a,o}$ se "contracte", couvre des plus hautes fréquences à une échelle plus "fine". Inversément, si $|a|$ est grand, $\psi^{a,o}$ se "dilate", couvrant des plus basses fréquences. Le paramètre b est le paramètre de translation en temps. En effet, $\psi^{a,b}$ est centrée autour de $t = b$. En conséquence, la Transformée en Ondelettes Continue fourni, comme la Transformée de Fourier Fenêtrée, une représentation temps-fréquence ! Cependant, comme indiqué sur la figure 1.10 (a) tirée de [6], les fonctions sur lesquelles on développe $x(t)$ de la Transformée de Fourier Fenêtrée ne sont que des enveloppes (les fenêtres) entourant une oscillation c'est-à-dire $win(t - t')e^{-i1\pi ft}$. Le problème étant, comme nous l'avons vu, que la largeur de l'enveloppe reste constante, quelque soit f ! L'avantage saillant de la Transformée en Ondelettes Continue est

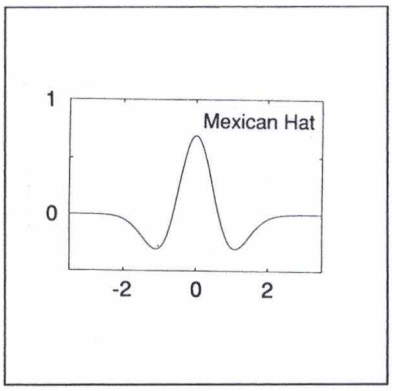


FIG. 1.9 – *Graphe du chapeau mexicain (Mexican Hat).*

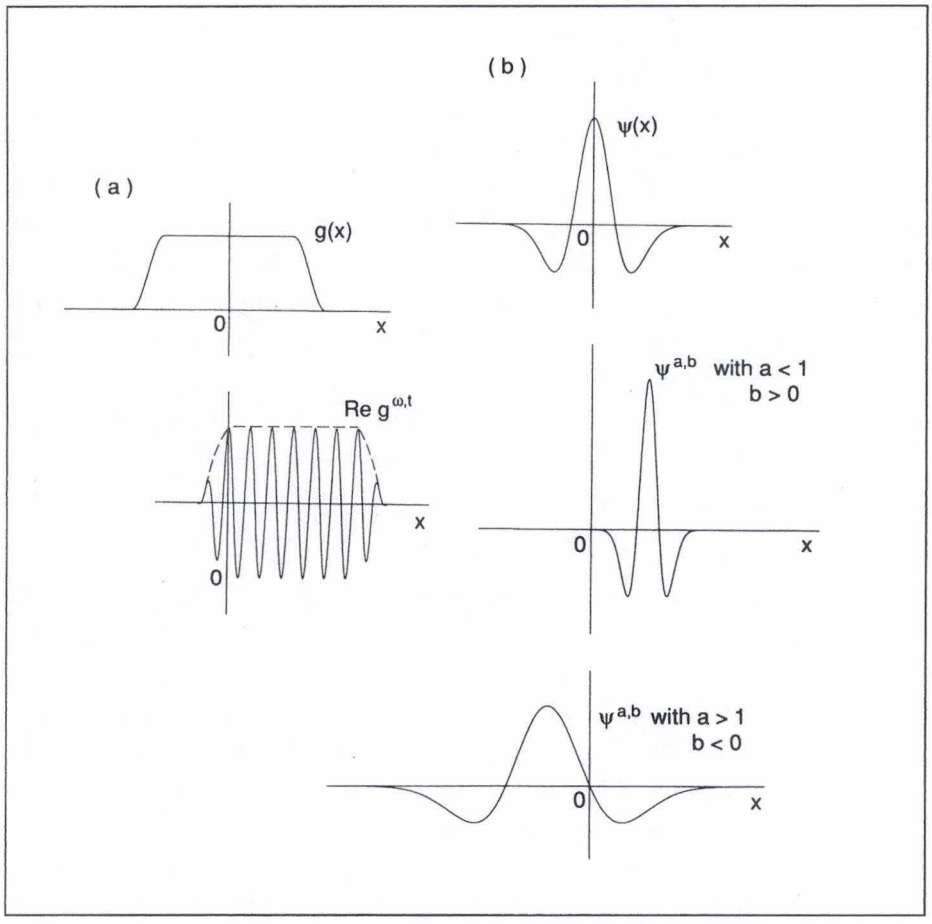


FIG. 1.10 – *(a)Fonctions sur lesquelles la Transformée de Fourier Fenêtrée développe $x(t)$. (b) Illustration des changements d'échelle et des translations du chapeau mexicain.*

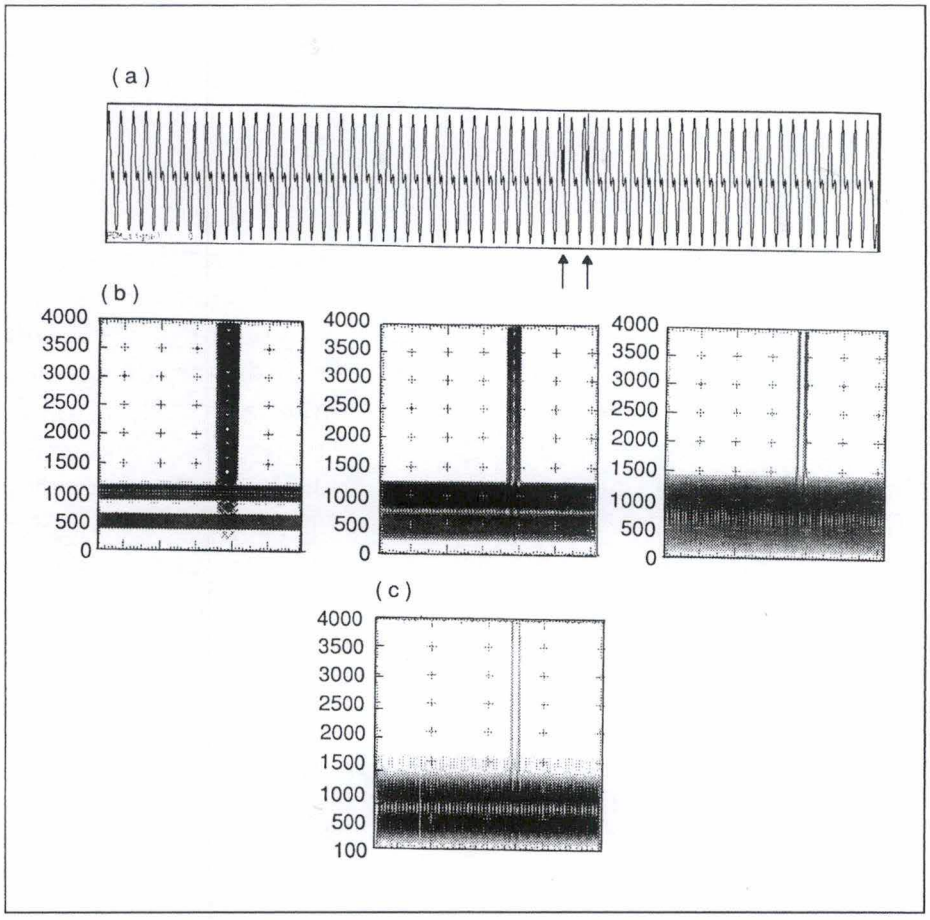


FIG. 1.11 – *(a) Le signal à analyser ; les deux flèches indiquent la présence des cusp. (b) Trois Tranformées de Fourier Fenêtrées du signal avec des fenêtres de largeur différentes. (c) La Transformée en Ondelettes du signal.*

que la largeur de $\psi^{a,b}$ en temps s'adapte à la fréquence : les $\psi^{a,b}$ correspondant à de basses fréquences sont larges et celles correspondant aux hautes fréquences sont étroites. Et ceci est particulièrement bien adapté à tous les signaux réels où les basses fréquences s'étendent sur la durée totale du signal qui présente quelques pics à hautes fréquences qui sont intéressants. La Transformée en Ondelettes Continue va s'adapter automatiquement à la détection de ces "*cusp*".

Dans [6], Daubechies nous donne un excellent exemple à titre de comparaison. Le signal de la figure 1.11 (a) est donné par :

$$x(t) = \sin(2\pi\nu_1 t) + \sin(2\pi\nu_2 t) + \gamma[\delta(t - t_1) + \delta(t - t_2)]$$

C'est-à-dire une composition de deux sinus présentant deux cusp (fonction delta)

en t_1 et t_2. Dans l'exemple, les paramètres sont :

$$\begin{aligned} \nu_1 &= 500\ Hz \\ \nu_2 &= 1\ kHz \\ \gamma &= 1.5 \\ t_2 - t_1 &= 4\ msec \end{aligned}$$

Les trois premiers graphes (représentation en niveaux de gris (inversés) du plan temps-fréquence) reprennent le module de la Transformée de Fourier Fenêtrée de ce signal calculée avec une fenêtre de Hamming (cfr figure 1.4) dont la largeur vaut 12.8, 6.4 et 3.2 msec respectivement. On peut s'apercevoir des problèmes déjà mentionnés pour la Transformée de Fourier Fenêtrée. Pour une large fenêtre, la résolution en fréquences est bonne, mais pas celle du temps : on ne peut pas distinguer les deux cusp. Inversément, une fenêtre plus étroite permet de bien voir les deux pics mais perd la résolution en fréquences : on n'arrive plus à distinguer les sinusoïdes. Le graphe suivant est la Transformée en Ondelettes Continue du même signal. On peut constater que les pics sont aussi bien définis en temps que pour la plus petite fenêtre de Hamming tout en conservant une résolution fréquentielle équivalente à la fenêtre de 6.4 msec pour les deux sinusoïdes.

1.2.2 Théorie de la Transformée en Ondelettes Continue

Nous allons donc examiner certains des résultats fondamentaux concernant la Transformée en Ondelettes Continue. Comme je l'ai déjà mentionné, mon propos n'est pas de réaliser un traité de mathématiques mais de permettre au lecteur de manipuler les principaux concepts des Ondelettes et ainsi de se familiariser avec cette matière. Tous les développements détaillés peuvent se trouver dans les références que j'ai déjà mentionnées.

Reprenons donc la définition de la Transformée en Ondelettes Continue.

$$(\mathcal{T}f)(a,b) = |a|^{-\frac{1}{2}} \int f(x)\ \overline{\psi\left(\frac{t-b}{a}\right)}\, \mathrm{d}x = \langle f, \psi^{a,b} \rangle$$

Nous avons vu qu'il s'agissait d'une décomposition de f sur une base d'Ondelettes formées de dilatations et de translations d'une Ondelette Mère : $\psi(t) \in \mathbf{L}^2(\mathbb{R})$.

La première chose que nous allons voir est que la Transformée en Ondelettes Continue est inversible : on peut reconstruire $f(t)$ à partir de sa décomposition. Pour ce faire, nous écrivons la formule suivante :

$$f = C_\psi^{-1} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} \frac{1}{a^2} \langle f, \psi^{a,b} \rangle \psi^{a,b}\ \mathrm{d}a\ \mathrm{d}b \tag{1.1}$$

$$\text{où } \psi^{a,b}(x) = |a|^{-\frac{1}{2}} \psi\left(\frac{x-b}{a}\right)\ ;\ a, b \in \mathbb{R},\ a \neq 0 \tag{1.2}$$

$$\text{et } C_\psi = 2\pi \int_{-\infty}^{+\infty} |\hat{\psi}(\epsilon)|^2 |\epsilon|^{-1} \mathrm{d}\epsilon < \infty \tag{1.3}$$

Cette dernière condition est appellée *condition d'admissibilité*. Elle est nécessaire pour que 1.1 ait du sens. Cette condition va nous permettre de donner une signification à $\int_{-\infty}^{+\infty} \psi(t)\mathrm{d}t = 0$ que nous avons rencontré tout à l'heure. En effet, si $\psi(x) \in \mathbf{L}^1(\mathbb{R})$ (comme c'est souvent le cas en pratique), on peut montrer que 1.3 est satisfaite si $\hat{\psi}(0) = 0$ ou si $\int \psi(x)\mathrm{d}x = 0$. D'autre part, si $\int \psi(x)\mathrm{d}x = 0$, on peut imposer à ψ de légères contraintes qui permettent de garantir 1.3. Il est donc de bonne guerre de considérer que 1.3 est équivalente à :

$$\int_{-\infty}^{+\infty} \psi(x)\mathrm{d}x = 0$$

Dans l'expression de 1.2, la présence de $|a|^{-\frac{1}{2}}$ nous assure que $\|\psi^{a,b}\| = \|\psi\|$. Par souci de simplification, nous considèrerons que $\|\psi\| = 1$.

On peut aussi prouver que $\forall f, g \in \mathbf{L}^2(\mathbb{R})$

$$\int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} (\mathcal{T}f)(a,b)\ \overline{(\mathcal{T}g)(a,b)}\ \frac{\mathrm{d}a\ \mathrm{d}b}{a^2} = C_\psi \langle f, g \rangle \tag{1.4}$$

On peut alors lire 1.4 comme une méthode de reconstruction de f :

$$f = C_\psi^{-1} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} (\mathcal{T}f)(a,b)\ \psi^{a,b}\ \frac{\mathrm{d}a\ \mathrm{d}b}{a^2} \tag{1.5}$$

Où la convergence de l'intégrale est assurée par le fait que prendre le produit scalaire de chacun des deux membres par une fonction quelconque $g \in \mathbf{L}^2(\mathbb{R})$ conduise à une formule vraie : 1.4. Daubechies montre dans [6] que l'on peut assurer la convergence de manière plus forte encore.

Il existe plusieurs variations de 1.5 dans lesquelles on restreint l'ensemble des valeurs de a aux réels positifs (dans 1.5, a peut également être négatif). On peut alors imposer à ψ une condition d'admissibilité plus stricte :

$$C_\psi = 2\pi \int_0^{+\infty} |\epsilon|^{-1}\ |\hat{\psi}(\epsilon)|^2\ \mathrm{d}\epsilon = 2\pi \int_{-\infty}^{0} |\epsilon|^{-1}\ |\hat{\psi}(\epsilon)|^2\ \mathrm{d}\epsilon < \infty$$

Cette condition est vérifiée si par exemple ψ est réelle puisqu'alors $\hat{\psi}(-\epsilon) = \overline{\hat{\psi}(\epsilon)}$. L'équation 1.5 devient alors :

$$f = C_\psi^{-1} \int_0^{\infty} \frac{\mathrm{d}a}{a^2} \int_{-\infty}^{+\infty} \mathrm{d}b\ (\mathcal{T}f)(a,b)\psi^{a,b}$$

avec le même type de convergence.

Si les supports de $\hat{f}$ et $\hat{\psi}$ sont inclus dans $[0, +\infty[$, alors $(\mathcal{T}f)(a,b) = 0$ et 1.5 se simplifie en

$$f = C_\psi^{-1} \int_0^{\infty} \frac{\mathrm{d}a}{a^2} \int_{-\infty}^{+\infty} \mathrm{d}b\ (\mathcal{T}f)(a,b)\psi^{a,b}$$

avec C_ψ définit comme 1.3.

Nous pouvons même introduire une seconde Ondelette, différente de la première et servant à la reconstitution. On aura que si ψ_1 et ψ_2 sont telles que :

$$\int |\epsilon|^{-1} |\hat{\psi}_1(\epsilon)| \, |\hat{\psi}_2(\epsilon)| \, < \, \infty$$

alors, on sait prouver que, $\forall f, g \in \mathbf{L}^2(\mathbb{R})$

$$\int \frac{\mathrm{d}a}{a^2} \int \mathrm{d}b \, \langle f, \psi_1^{a,b} \rangle \, \langle \psi_2^{a,b}, g \rangle \, = \, C_{\psi_1,\psi_2} \langle f, g \rangle$$

$$\text{avec } C_{\psi_1,\psi_2} \, = \, 2\pi \int |\epsilon|^{-1} \, \overline{\hat{\psi}_1(\epsilon)} \, \hat{\psi}_2(\epsilon) \, \mathrm{d}\epsilon$$

Si $C_{\psi_1,\psi_2} \neq 0$, on peut écrire la variante de 1.5 obtenue :

$$f \, = \, C_{\psi_1,\psi_2}^{-1} \int \frac{\mathrm{d}a}{a^2} \int \mathrm{d}b \langle f, \psi_1^{a,b} \rangle \, \psi_2^{a,b} \tag{1.6}$$

Remarquons que ψ_1 et ψ_2 peuvent être très différentes, elles ne sont même pas obligatoirement admissibles toutes les deux. On trouvera dans [6] une utilisation intéressante de la liberté dans le choix de ces deux fonctions.

Intéressons-nous à présent à l'exécution de la Transformée en Ondelettes Continue à plusieurs dimensions. Il existe en fait plusieurs possibilités pour faire ceci. Remarquons qu'il n'existe toujours qu'un seul paramètre d'échelle a mais que les déplacements (qui ne sont pas seulement des translations) se font dans un espace à deux dimensions au moins. Nous considèrerons également que $\psi \in \mathbf{L}^2(\mathbb{R}^n)$ avec $n > 1$.

Ainsi, nous pouvons imposer que ψ possède une symétrie sphérique. On traduit cela par l'existence d'une fonction τ telle que :

$$\psi(x) \, = \, \tau(|x|) \quad \forall x \in \mathbb{R}^n$$

Il existe donc aussi une fonction η telle que :

$$\hat{\psi}(x) \, = \, \eta(|x|)$$

La condition d'admissibilité devient

$$C_\psi \, = \, (2\pi)^n \int_0^\infty \frac{\mathrm{d}t}{t} |\eta(t)|^2 < \infty$$

On généralise 1.5 par :

$$f \, = \, C_\psi^{-1} \int_0^\infty \frac{\mathrm{d}a}{a^{n+1}} \int_{\mathbb{R}^n} \mathrm{d}b \, (\mathcal{T}f)(a,b) \psi^{a,b}$$

Il est également possible de choisir une fonction ψ qui ne soit pas de symétrie sphérique. On peut alors introduire des rotations et des translations. Par exemple, à deux dimensions :

$$\psi^{a,b,\theta}(x) \;=\; a^{-1}\,\psi\left(\mathbf{R}_\theta^{-1}\left(\frac{x-b}{a}\right)\right)$$

avec $a > 0, b \in \mathbb{R}^2$ et où $\mathbf{R}_\theta$ est une matrice de rotation, soit :

$$\mathbf{R}_\theta \;=\; \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

La condition d'admissibilité devient :

$$C_\psi \;=\; (2\pi)^2 \int_0^\infty \frac{\mathrm{d}r}{r} \int_0^{2\pi} \mathrm{d}\theta\; |\hat{\psi}(r\cos\theta, r\sin\theta)|^2 \;<\infty$$

et 1.5 se transforme en :

$$f \;=\; C_\psi^{-1} \int_0^\infty \frac{\mathrm{d}a}{a^3} \int_{\mathbb{R}^2} \mathrm{d}b \int_0^{2\pi} \mathrm{d}\theta\; (\mathcal{T}f)(a,b,\theta)\psi^{a,b,\theta}$$

Enfin, revenons un instant sur la Transformée de Fourier Fenêtrée et faisons un court parallèle entre celle-ci et la Transformée en Ondelettes Continue. En effet, la Transformée de Fourier Fenêtrée de $f \in \mathbf{L}^2(\mathbb{R})$ est définie par :

$$(\mathcal{F}^{win}f)(\omega,t) \;=\; \langle f, g^{\omega,t}\rangle$$

où $g^{\omega,t}(x) \;=\; e^{i\omega x} g(x-t)$
et $g(x)$ est une fonction de fenêtrage

On voit immédiatement la similitude entre les Ondelettes et $g^{w,t}$. D'ailleurs, de la même manière que plus haut, on peut prouver que $\forall f_1, f_2 \in \mathbf{L}^2(\mathbb{R})$

$$\int\int \mathrm{d}\omega\; \mathrm{d}t\; (\mathcal{F}^{win}f_1)(\omega,t)\; \overline{(\mathcal{F}^{win}f_2)(\omega,t)} \;=\; 2\pi\|g\|^2\langle f_1, f_2\rangle$$

Ce que l'on traduira par

$$f \;=\; (2\pi\|g\|^2)^{-1} \int\int \mathrm{d}\omega\; \mathrm{d}t\; (\mathcal{F}^{win}f)(\omega,t) g^{\omega,t}$$

Ce qui nous donne une relation de reconstruction de f à partir de sa Transformée de Fourier Fenêtrée.

Les différences que nous avons déjà mentionnées existent toujours : la largeur de la fenêtre est toujours fixée et ne change pas avec la fréquence. Cependant, il en est une nouvelle, plus inattendue, qui est qu'il n'existe pas de condition d'admissibilité : toute fonction g de $\mathbf{L}^2(\mathbb{R})$ fait l'affaire ! On choisi toutefois généralement g telle que $\|g\| \;=\; 1$. L'absence de cette condition d'admissibilité n'est absolument pas triviale et a déjà fait l'objet de plusieurs études.

Chapitre 2

L'Analyse multirésolution et la Transformée en Ondelettes Discrète

La rapidité de traitement des images par le cerveau humain reste un sujet d'étonnement à l'heure actuelle. En effet, nos neurones fonctionnent à des vitesses faibles (~100 msec) et le cerveau peut, en quelques secondes, prendre des décisions sur base d'images de plusieurs millions de points. Sachant que souvent les caractéristiques saillantes d'une image apparaissent déjà à faible résolution, on est tenté d'expliquer cette efficacité par une forme de traitement multirésolution, hiérarchique, en quelques couches. Le cerveau décomposerait l'image en plusieurs niveaux de résolutions qu'il traiterait rapidement. Sur base de cette idée, plusieurs approches se sont construites pour analyser les images. Elles ont donné naissance aux algorithmes pyramidaux [30, 5]. En 1989, sur base des travaux de Meyer et Lemarié [26, 20, 31], Stéphane Mallat faisait le lien entre ces approches et les ondelettes pour donner naissance à l'analyse multirésolution [25]. Cet outil formidable a été très vite raffiné et amélioré pour constituer aujourd'hui l'algorithme de la Transformée en Ondelettes Discrète qui est utilisé dans la plupart des applications informatiques de la Transformée en Ondelettes.

Dans ce chapitre, j'ai choisi de suivre Mallat dans son article fondateur en y insérant quelques-unes des améliorations ajoutées à son travail. Le lecteur intéressé pourra trouver tous les renseignements possibles dans [25, 15] [6, chapitres 5 et 6], [48, 8]. Encore une fois, toutes les notations et notions de base sont définies dans l'annexe A.

2.1 Approximation Multiresolution dans $L^2(\mathbb{R})$

Nous allons tenter de décrire les images dans une décomposition multirésolution. En d'autres termes, nous allons approximer l'image à différentes résolutions : aux

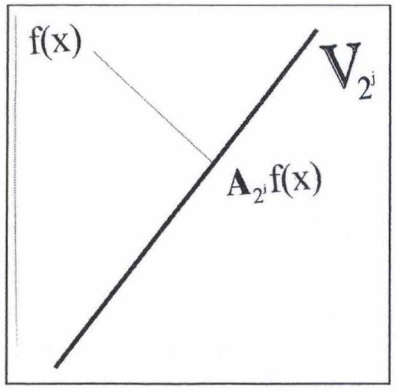


FIG. 2.1 – *Une projection orthogonale.*

résolutions basses, les "détails" de l'image correspondent à de "grandes" structures et aux résolutions élevées, on peut observer des structures beaucoup plus petites. Pour une suite de résolution $(r_j)_{j\in\mathbb{Z}}$, les détails d'une image à la résolution r_j sont définis comme la différence d'information entre l'approximation à la résolution r_j et l'approximation à la résolution r_{j+1}. Il paraît logique d'analyser d'abord l'image à basse résolution pour ensuite augmenter graduellement celle-ci. En règle générale, l'augmentation de résolution se fait par pas de deux, sous forme d'une progression géométrique telle que $r_j = 2^j$. Les raisons de ce choix apparaîtront plus tard.

Dans un premier temps, nous allons définir et étudier un opérateur qui transforme un signal en une approximation à la résolution de 2^j. Nous étudierons le cas à une dimension pour l'étendre ensuite à deux dimensions.

Soit $f(x) \in \mathbf{L}^2(\mathbb{R})$ et $\mathbf{A}_{2^j}$ l'opérateur qui approxime le signal à la résolution 2^j. Mallat définit six propriétés fondamentales que doit respecter cet opérateur. Ces propriétés sont les traductions des propriétés intuitives d'un tel opérateur.

1. $\mathbf{A}_{2^j}$ est une projection, c'est-à-dire que si on approxime une fonction à la résolution 2^j $(\mathbf{A}_{2^j} f(x))$, la réapproximation à la résolution 2^j n'a aucun effet. En d'autres termes : $\mathbf{A}_{2^j} \circ \mathbf{A}_{2^j} = \mathbf{A}_{2^j}$, ce qui est une caractéristique des opérateurs de projection. On définira donc l'espace vectoriel $\mathbb{V}_{2^j} \subset \mathbf{L}^2(\mathbb{R})$ tel que $\mathbf{A}_{2^j} f(x) \in \mathbb{V}_{2^j}$. On peut alors interpréter $\mathbb{V}_{2^j}$ comme l'ensemble de toutes les approximations possibles à la résolution de 2^j des fonctions de $\mathbf{L}^2(\mathbb{R})$.
2. $$\forall g(x) \in \mathbb{V}_{2^j}, \|g(x) - f(x)\| \geqslant \|\mathbf{A}_{2^j} f(x) - f(x)\| \tag{2.1}$$

   Soit $\mathbf{A}_{2^j} f(x)$ est la meilleure approximation de $f(x)$ à la résolution 2^j, c'est-à-dire qu'il n'existe pas d'approximation à la résolution 2^j qui soit plus proche de $f(x)$ que $\mathbf{A}_{2^j} f(x)$. On traduira cette propriété par le fait qu'$\mathbf{A}_{2^j} f(x)$ est une projection orthogonale sur $\mathbb{V}_{2^j}$ comme illustré à la figure 2.1 .

3.

$$\forall j \in \mathbb{Z} : \mathbb{V}_{2^j} \subset \mathbb{V}_{2^{j+1}} \tag{2.2}$$

Ceci est la traduction d'un principe de causalité : à partir d'une approximation à haute résolution, je peux reconstituer une approximation à résolution plus faible. Une approximation du signal à la résolution 2^{j+1} contient toute l'information pour constituer une approximation à la résolution 2^j.

4.

$$\forall j \in \mathbb{Z} \quad : f(x) \in \mathbb{V}_{2^j} \Leftrightarrow f(2x) \in \mathbb{V}_{2^{j+1}} \tag{2.3}$$

Cette propriété traduit simplement le fait que chaque espace de fonction peut être dérivé d'un autre par une transformation d'échelle sur ses éléments.

5. $\mathbf{A}_{2^j} f(x)$ peut être caractérisé par 2^j échantillons par unité de longeur. Lorsque $f(x)$ est translaté de $k2^{-j} (k \in \mathbb{Z})$, $\mathbf{A}_{2^j} f(x)$ est translaté de la même longeur et caractérisé par le même échantillonage translaté également. On traduira ceci par :

   a Caractérisation discrète :

$$\exists \text{ un isomorphisme } \mathbf{I} \text{ entre } \mathbb{V}_1 \text{ et } \mathbf{l}^2(\mathbb{Z}) \tag{2.4}$$

   b Translation :

$$\forall k \in \mathbb{Z},\ \mathbf{A}_1 f_k(x) = \mathbf{A}_1 f(x-k) \text{ où } f_k(x) = f(x-k) \tag{2.5}$$

   c Translation de l'échantillonnage :

$$\mathbf{I}(\mathbf{A}_1 f(x)) = (\alpha_i)_{i\in\mathbb{Z}} \Leftrightarrow \mathbf{I}(\mathbf{A}_1 f_k(x)) = (\alpha_{i-k})_{i\in\mathbb{Z}} \tag{2.6}$$

   On étend ces propriétés aux espaces $\mathbb{V}_{2^j}$ grâce à 2.3

6. Approximer représente une perte d'information. Pour des résolutions tendant vers $+\infty$, l'approximation tend vers le signal original et inversément, pour des résolutions tendant vers $-\infty$, l'approximation tend vers zéro, c'est-à-dire aucune information.
   On réécrit ceci par :

$$\lim_{j\to+\infty} \mathbb{V}_{2^j} = \bigcup_{j=-\infty}^{+\infty} \mathbb{V}_{2^j} \text{ est dense dans } \mathbf{L}^2(\mathbb{R}) \tag{2.7}$$

$$\text{et } \lim_{j\to-\infty} \mathbb{V}_{2^j} = \bigcap_{j=-\infty}^{+\infty} \mathbb{V}_{2^j} = \{0\} \tag{2.8}$$

Mallat appelle tout ensemble d'espace vectoriel $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ satisfaisant les propriétés 2.2 - 2.8 une *Approximation Multirésolution* de $\mathbf{L}^2(\mathbb{R})$. L'ensemble des opérateurs $\mathbf{A}_{2^j}$ satisfaisant 2.1 - 2.6 donnent les approximations des fonctions de $\mathbf{L}^2(\mathbb{R})$ à la résolution 2^j.

Donnons un exemple simple d'une telle approximation. Soit $\mathbb{V}_1$ l'ensemble de toutes les fonctions constantes sur tout intervalle $]k, k+1]\ \forall k \in \mathbb{Z}$. Par 2.3, on sait que $\mathbb{V}_{2^j}$ est l'espace des fonctions constantes sur tout intervalle $]k2^{-j},\ (k+1)2^{-j}]\ \forall k \in \mathbb{Z}$. 2.2 est évidemment vérifiée dans ces conditions. L'isomorphisme $\mathbf{I}$ satisfaisant 2.4, 2.5 et 2.6 se définit en associant à $f(x) \in \mathbb{V}_1$ la série $(\alpha_k)_{k\in\mathbb{Z}}$ avec α la valeur de $f(x)$ sur l'intervalle $]k, k+1]$. On peut prouver que l'ensemble des fonctions constantes par intervalle est dense dans $\mathbf{L}^2(\mathbb{R})$, donc, par extension, $\bigcup_{j=-\infty}^{+\infty} \mathbb{V}_{2^j}$ est également dense dans $\mathbf{L}^2(\mathbb{R})$. On peut voir facilement que $\bigcap_{j=-\infty}^{+\infty} \mathbb{V}_{2^j} = \{0\}$. Donc, $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ est une approximation multirésolution de $\mathbf{L}^2(\mathbb{R})$. Malheureusement, cette approximation, bien que très simple, est peu utile à cause de sa discontinuité.

Un premier théorème important de l'analyse multirésolution est celui qui va nous permettre de caractériser l'opérateur $\mathbf{A}_{2^j}$. Pour ce faire, il faut trouver une base orthonormale de $\mathbb{V}_{2^j}$ puisque $\mathbf{A}_{2^j}$ est une projection sur cet espace.

Théorème 1

Soit $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ une approximation multirésolution de $\mathbf{L}^2(\mathbb{R})$. Il existe une fonction unique $\phi(x) \in \mathbf{L}^2(\mathbb{R})$ telle que si on pose $\phi_{2^j}(x) = 2^j\phi(2^j x)\ \forall j \in \mathbb{Z}$ (c'est-à-dire que $\phi_{2^j}(x)$ est une simple dilatation par 2^j de $\phi(x)$), alors $(\sqrt{2^{-j}}\phi_{2^j}(x - 2^{-j}n))_{n\in\mathbb{Z}}$ est une base orthonormale de $\mathbb{V}_{2^j}$.

■

La preuve de ce théorème peut être touvée dans [24].

Donc, pour une Approximation multirésolution, on peut bâtir une base orthonormale pour chaque ensemble $\mathbb{V}_{2^j}$ par dilatation d'une fonction $\phi(x)$ d'un coefficient 2^j et translation par pas de 2^{-j}. Le coefficient $\sqrt{2^{-j}}$ est dû à la normalisation par rapport à $\mathbf{L}^2(\mathbb{R})$. Il est clair que les fonctions $\phi(x)$ sont attachées à des Approximations Multirésolutions données, des Approximations Multirésolutions différentes ont des fonctions $\phi(x)$ différentes. On nommera $\phi(x)$ la fonction d'échelle (scaling function). Pour l'exemple ci-dessus, $\phi(x)$ est la fonction $= 1$ sur l'intervalle $[0, 1]$ et nulle partout ailleurs. Cette fonction n'est pas très intéressante à cause de sa discontinuité. Les applications pratiques demandent d'autres propriétés. Mallat donne un exemple de fonction continuement différentiable et exponentiellement décroissante générant une Approximation Multirésolution. Il s'agit de la fonction de Battle-Lemarié dont la Transformée de Fourier montre qu'il s'agit d'un filtre passe-bas (figure 2.2). On trouvera une description plus poussée de cette fonction dans l'annexe B.

A présent, nous pouvons exprimer l'effet de l'opérateur $\mathbf{A}_{2^j}$.

$$\forall f(x)\ \in \mathbf{L}^2(\mathbb{R})\ : \mathbf{A}_{2^j} f(x)\ =\ 2^{-j} \sum_{n=-\infty}^{+\infty} \langle f(u), \phi_{2^j}(u - 2^{-j}n)\rangle\ \ \phi_{2^j}(x - 2^{-j}n)$$

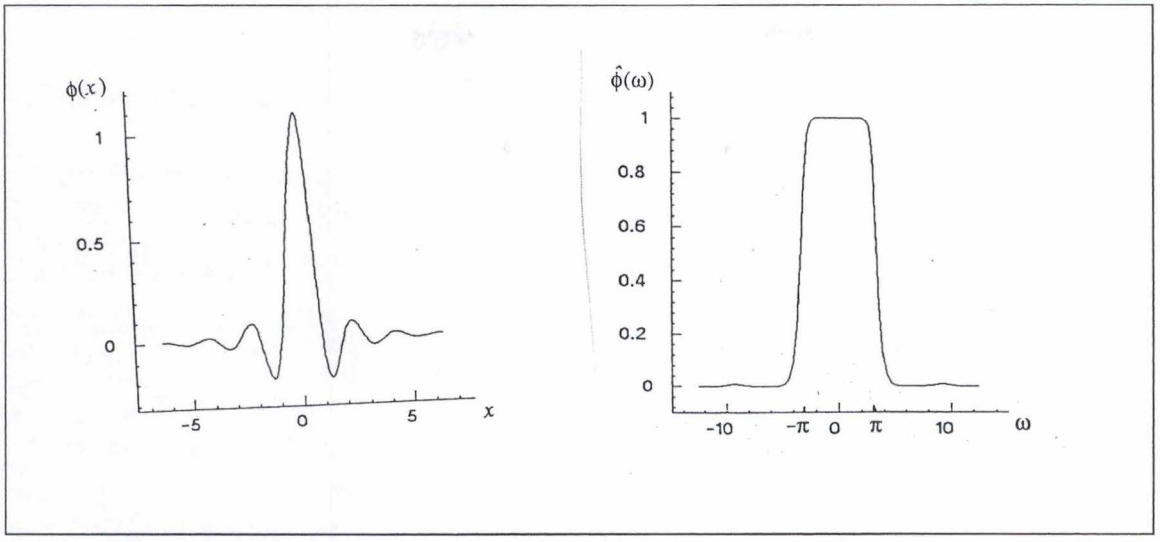


FIG. 2.2 – *Un exemple de fonction d'échelle $\phi(x)$ et sa Transformée de Fourier $\hat{\phi}(x)$.*

Ce qui exprime simplement la projection de $f(x)$ sur la base définie par le théorème 1. L'ensemble des coefficients qui caractérisent cette décomposition est notée par Mallat $\mathbf{A}_{2^j}^d f$ avec

$$\mathbf{A}_{2^j}^d f = \left(\langle f(u),\ \phi_{2^j}(u - 2^{-j}n)\rangle\right)_{n\in\mathbb{Z}}$$

et appellée *approximation discrète* de $f(x)$ à la résolution 2^j.

Réécrivons chacun des produits scalaires :

$$\begin{aligned}\langle f(u),\ \phi_{2^j}(u - 2^{-j}n)\rangle &= \int_{-\infty}^{+\infty} f(u)\phi_{2^j}(u - 2^{-j}n)\mathrm{d}u \\ &= (f(u) * \phi_{2^j}(-u))(-2^{-j}n) \\ &= (f(u) * \phi_{2^j}(-u))(2^{-j}(-n))\end{aligned}$$

Comme n parcourt tout l'ensemble $\mathbb{Z}$, on peut écrire :

$$\mathbf{A}_{2^j}^d f = \left((f(u) * \phi_{2^j}(-u))(2^{-j}n)\right)_{n\in\mathbb{Z}}$$

Comment interpréter cette formule ? Rappelons qu'une convolution en temps correspond à une multiplication en fréquence (cfr. annexe A). Donc, $\mathbf{A}_{2^j}^d f$ correspond à la foncion f passée par un filtre passe-bas et échantillonné à la fréquence 2^j. Ceci paraît bien logique : en approximant la fonction, on perd les détails que constituent les hautes fréquences. De plus, comme le fait très justement remarquer Mallat, $\phi(u)$ n'est pas un simple filtre passe-bas car la famille $(\sqrt{2^{-j}}\phi_{2^j}(x - 2^{-j}x))_{x\in\mathbb{Z}}$ forme une base orthonormale de l'espace $\mathbb{V}_{2^j}$ ce qui constitue une propriété remarquable.

2.2 Transformation Multirésolution

Ce paragraphe va décrire le remarquable algorithme proposé par Mallat pour implémenter la Transformation Multirésolution.

Considérons un signal physique échantillonné. Pour des raisons de simplicité, nous considèrerons que ce signal possède une résolution de 1, il constitue donc $\mathbf{A}_1^d f$ (soit $\mathbf{A}_{2^j}^d f$ avec $j = 0$). Ceci paraît logique : le système physique a approximé un signal f avec une résolution de 1.

Nous savons par la propriété 3 que nous pouvons reconstruire toute approximation $\mathbf{A}_{2^j}^d f$ si $j < 0$: c'est cette opération que nous allons implémenter.

Soit $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ une Approximation Multirésolution de $\mathbf{L}^2(\mathbb{R})$ et $\phi(x)$ la fonction d'échelle associée. On sait que $\forall n \in \mathbb{Z}$, $\phi_{2^j}(x - 2^{-j}n) \in \mathbb{V}_{2^j}$ dont elle est un élément d'une base orthonormale. Or, $\mathbb{V}_{2^j}$ est inclus dans $\mathbb{V}_{2^{j+1}}$. Donc, on peut exprimer $\phi_{2^j}(x - 2^{-j}n)$ dans une base de $\mathbb{V}_{2^{j+1}}$, la base donnée par le théorème 1, on écrit :

$$\phi_{2^j}(x - 2^{-j}n) \;=\; 2^{-j-1} \sum_{k=-\infty}^{+\infty} \langle .\phi_{2^j}(u - 2^{-j}n),\ \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle . \phi_{2^{j+1}}(x - 2^{-j-1}k)$$

Un rapide calcul va nous permettre de mettre en évidence une propriété remarquable du produit scalaire dans cette somme. En effet,

$$2^{-j-1}\langle \phi_{2^j}(u - 2^{-j}n),\ \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle =$$

$$2^{-j-1} \int 2^j \phi(2^j u - n)\ 2^{j+1}\phi(2^{j+1}u - k)\mathrm{d}u$$

Si on pose $u' = 2(2^j u - n)$, on a $\mathrm{d}u' = 2^{j+1}\mathrm{d}u$ et $\mathrm{d}u = 2^{-j-1}\mathrm{d}u'$ et donc $u = 2^{-j-1}(u' - 2n)$.

En effectuant le changement de variable, on obtient :

$$\int 2^{-1}\phi(2^{-1}u')\phi(u' - (k - 2n))\mathrm{d}u' \;=\; \langle \phi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle$$

Nous pouvons donc réécrire :

$$\phi_{2^j}(x - 2^{-j}n) = \sum_{k=-\infty}^{+\infty} \langle \phi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle \phi_{2^{j+1}}(x - 2^{-j-1}k)$$

En prenant le produit scalaire des deux membres avec $f(u)$, on obtient :

$$\langle f(u),\ \phi_{2^j}(u - 2^{-j}n)\rangle \;=\; \sum_{k=-\infty}^{+\infty} \langle \phi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle \langle f(u), \phi_{2^{j+1}}(x - 2^{-j-1}k)\rangle$$

Soit $\mathbf{H}$ un filtre discret dont la réponse impulsionnelle serait donnée par

$$\forall n \in \mathbb{Z},\ h(n) \;=\; \langle \phi_{2^{-1}}(u),\ \phi(u - n)\rangle$$

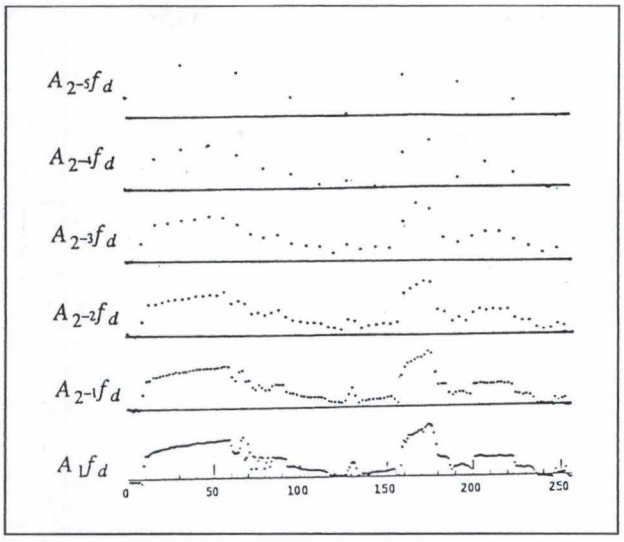


FIG. 2.3 – *Les approximations discrètes* $\mathbf{A}_{2^j}^d f$ *pour* $0 \leqslant j \leqslant 5$. *Chaque point correspond à un produit scalaire* $\langle f(u), \phi_{2^j}(u - 2^{-j}n)\rangle$.

Soit $\tilde{\mathbf{H}}$ tel que $\tilde{h}(n) = h(-n)$
On peut réécrire l'équation précédente par :

$$\langle f(u),\ \phi_{2^j}(u - 2^{-j}n)\rangle \ = \ \sum_{k=-\infty}^{+\infty} \tilde{h}(2n-k)\ \langle f(u), \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle$$

On a donc :

$$(\mathbf{A}_{2^j}^d f)(n) = (\mathbf{A}_{2^{j+1}}^d f * \tilde{h})(2n)$$

Ainsi, on peut obtenir $\mathbf{A}_{2^j}^d f$ en convoluant $\mathbf{A}_{2^{j+1}}^d f$ avec $\tilde{\mathbf{H}}$ et en ne gardant qu'un échantillon sur deux. On peut donc obtenir toutes les approximations $\mathbf{A}_{2^j}^d f\ \forall j \in \mathbb{Z},\ j < 0$ en répétant cette opération autant de fois qu'il le faut à partir de $\mathbf{A}_1^d f$. On appelle cela un algorithme pyramidal. La figure 2.3 , tirée de [25] illustre cette procédure.

Il est un problème qui se présente évidemment lorsque l'on veut effectuer les calculs, qui est celui du nombre fini d'échantillons.

On a $\mathbf{A}_1^d f = (\alpha_i)_{0 \leqslant i \leqslant N}$. Pour éviter les problèmes avec frontières, on considèrera que $\mathbf{A}_1^d f$ est symétrique par rapport à $i = 0$ et $i = N$. C'est-à-dire :

$$\alpha_n = \begin{cases} \alpha_{-n} & \text{si } -N < n < 0 \\ \alpha_{2N-n} & \text{si } N < n < 2N \end{cases}$$

Alors que le théorème 1 nous assure de l'existence d'une base orthonormale pour une Approximation Multirésolution donnée, il ne nous donne aucune indication sur

la fonction d'échelle $\phi(x)$ dont les dilatations et translations forment cette base. A la suite de Mallat, imposons deux conditions supplémentaires à $\phi(x)$ pour pouvoir proposer un théorème quant à la caractérisation de la Transformée de Fourier de cette fonction. On pose donc que $\phi(x)$ doit être continuement différentiable et que

$$|\phi(x)| = \mathbf{O}(x^{-2}) \text{ et } \left|\frac{\mathrm{d}\phi(x)}{\mathrm{d}x}\right| = \mathbf{O}(x^{-2}) \text{ à l'infini}$$

Nous sommes prêts pour le théorème 2.

Théorème 2

Soit $\phi(x)$, une fonction, d'échelle , soit $\mathbf{H}$ un filtre dont la réponse impulsionelle est $h(x) = \langle \phi_{2^{-1}}(u),\ \phi(u-n)\rangle$. Posons que $\mathbf{H}(\omega)$ ait la série de Fourier définie par :

$$\mathbf{H}(\omega) = \sum_{n=-\infty}^{+\infty} h(n)e^{-in\omega}$$

Alors, $\mathbf{H}(\omega)$ satisfait les propriétés suivantes :

$$|\mathbf{H}(0)| = 1 \text{ et } h(n) = \mathbf{O}(n^{-2}) \text{ à l'infini}$$
$$|\mathbf{H}(\omega)|^2 + |\mathbf{H}(\omega+\pi)|^2 = 1$$

Si on a de plus que

$$|\mathbf{H}(\omega)| \neq 0 \text{ avec } \omega \in \left[0, \frac{\pi}{2}\right]$$

Alors,

$$\hat{\phi}(\omega) = \prod_{p=1}^{+\infty} \mathbf{H}(2^{-p}\omega) \tag{2.9}$$

est la Transformée de Fourier de $\phi(x)$

■

La preuve de ce théorème peut se trouver dans [24] .

Les filtres satisfaisant les propriétés de ce théorème ont été abondamment étudiés, on les appelle filtres conjugés. Lorsqu'on connaît un filtre conjugué, il est facile, à présent, de calculer la fonction d'échelle grâce à sa Transformée de Fourier donnée par 2.9. Il est clair que le choix de $\mathbf{H}(\omega)$ (et donc de $h(n)$) va influencer $\phi(x)$. Si on reprend l'exemple sus-cité, on peut montrer que :

$$\mathbf{H}(\omega) = e^{-i\frac{\omega}{2}}\cos(\frac{\omega}{2})$$

2.3.1 Calculer les détails du signal

Rappellons que les détails du signal à la résolution 2^j sont définis comme la différence d'information entre les approximations de $f(x)$ à la résolution 2^{j+1} et à la résolution 2^j. Ces approximations sont données par les projections de $f(x)$ sur les espaces vectoriels $\mathbb{V}_{2^{j+1}}$ et $\mathbb{V}_{2^j}$ respectivement. Sachant que $\mathbb{V}_{2^j} \subset \mathbb{V}_{2^{j+1}}$, il est clair que les détails du signal à la résolution 2^j sont donnés par la projection de $f(x)$ sur le complément orthogonal de $\mathbb{V}_{2^j}$ dans $\mathbb{V}_{2^{j+1}}$. Posons $\mathbb{O}_{2^j}$ ce complément, on a que :

$$\begin{array}{l} \mathbb{O}_{2^j} \text{ est orthogonal à } \mathbb{V}_{2^j} \\ \mathbb{O}_{2^j} \oplus \mathbb{V}_{2^j} \ = \ \mathbb{V}_{2^{j+1}} \end{array}$$

Le théorème suivant nous donne les moyens de trouver une base orthonormale pour $\mathbb{O}_{2^j}$ de la même manière que le théorème 1 nous permettait de trouver une base orthonormale de $\mathbb{V}_{2^j}$.

Théorème 3

Soit $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ une Approximation Multirésolution de $\mathbf{L}^2(\mathbb{R})$ et $\phi(x)$ sa fonction d'échelle dont $\mathbf{H}$ est le filtre conjugué.

Soit $\psi(x)$ une fonction dont la Transformée de Fourier est donnée par :

$$\hat{\psi}(\omega) = \mathbf{G}(\frac{\omega}{2})\ \hat{\phi}(\frac{\omega}{2}) \qquad \text{avec } \mathbf{G} \ = \ e^{-i\omega}\ \overline{\mathbf{H}(\omega+\pi)}$$

Comme pour $\phi(x)$, posons $\psi_{2^j} = 2^j\psi(2^j x)$. Alors, nous avons que

$$(\sqrt{2^{-j}}\ \psi_{2^j}(x - 2^{-j}n))_{n\in\mathbb{Z}}$$

est une base orthonormale de $\mathbb{O}_{2^j}$ et

$$(\sqrt{2^{-j}}\ \psi_{2^j}(x - 2^{-j}n))_{(n,j)\in\mathbb{Z}^2}$$

est une base orthonormale de $\mathbf{L}^2(\mathbb{R})$.

On appelle $\psi(x)$ une Ondelette orthogonale.

■

La preuve de ce théorème peut être trouvée, comme toutes les autres, dans [24].

Les conséquences de ce théorème sont très importantes. Il nous permet de construire une base orthonormale de $\mathbb{O}_{2^j}$ à partir d'une Ondelette dont on connaît la Transformée de Fourier. Cette base servira à exprimer les détails du signal et sera donc la pierre d'angle de notre représentation. Si nous continuons sur notre exemple simple, l'Ondelette $\psi(x)$ correspondante est l'Ondelette de Haar, définie par

$$\psi(x) = \begin{cases} 1 & \text{si } 0 \leqslant x < \frac{1}{2} \\ -1 & \text{si } \frac{1}{2} \leqslant x < 1 \\ 0 & \text{sinon} \end{cases}$$

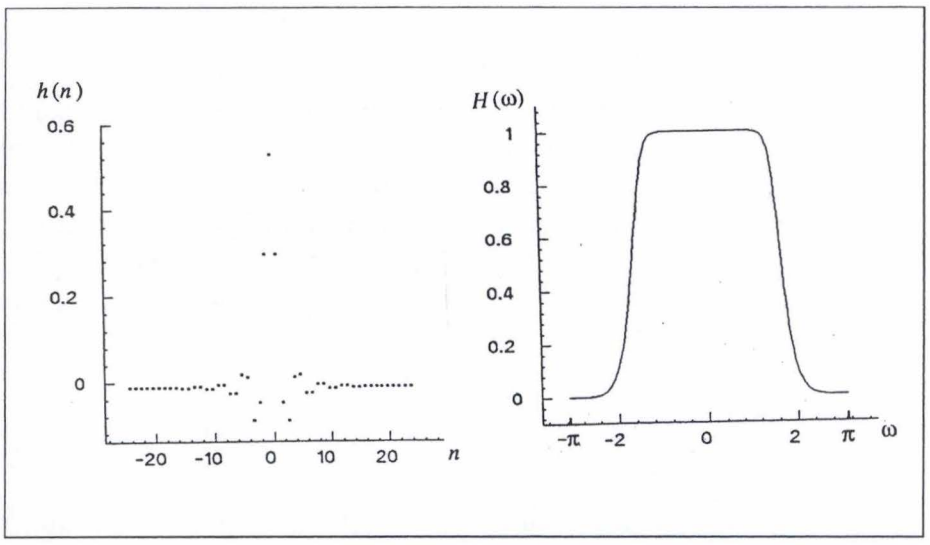


FIG. 2.4 – *(a) Réponse implusionnelle du filtre* **H** *associé à la fonction d'échelle de la figure 2.2 (b)* $\mathbf{H}(\omega)$ *pour ce filtre.*

En effet,

$$\begin{aligned} h(n) &= \langle \phi_{2^{-1}}(u),\ \phi(u-n) \rangle \\ &= \int_{-\infty}^{+\infty} \phi_{2^{-1}}(u)\phi(u-n)\mathrm{d}u \\ &= \tfrac{1}{2}\int_{0}^{2} \phi(u-n)\mathrm{d}u \end{aligned}$$

Donc seuls deux termes seront non nuls : $h(0) = h(1) = \frac{1}{2}$

$$\begin{aligned} \mathbf{H}(\omega) = \tfrac{1}{2} + \tfrac{e^{-i\omega}}{2} &= \tfrac{1}{2}(1 + \cos\omega - i\sin\omega) \\ &= \tfrac{1}{2}(2\cos^2\tfrac{\omega}{2} - 2i\sin\tfrac{\omega}{2}\cos\tfrac{\omega}{2}) \\ &= \cos\tfrac{\omega}{2}(\cos\tfrac{\omega}{2} - i\sin\tfrac{\omega}{2}) \\ &= \cos\tfrac{\omega}{2}e^{-i\frac{\omega}{2}} \end{aligned}$$

Toujours dans l'annexe B, on trouvera la suite de la description de la fonction de Battle-Lemarié dont la figure 2.4 nous donne le filtre **H**.

2.3 Représentation en ondelettes

Nous arrivons au coeur de la théorie. Notre but est de construire une représentation multirésolution basée sur les différences d'information qui existent entre deux approximations successives aux résolutions 2^j et 2^{j+1}. Nous allons montrer que cette représentation peut être calculée à partir d'une base orthonormale d'Ondelettes.

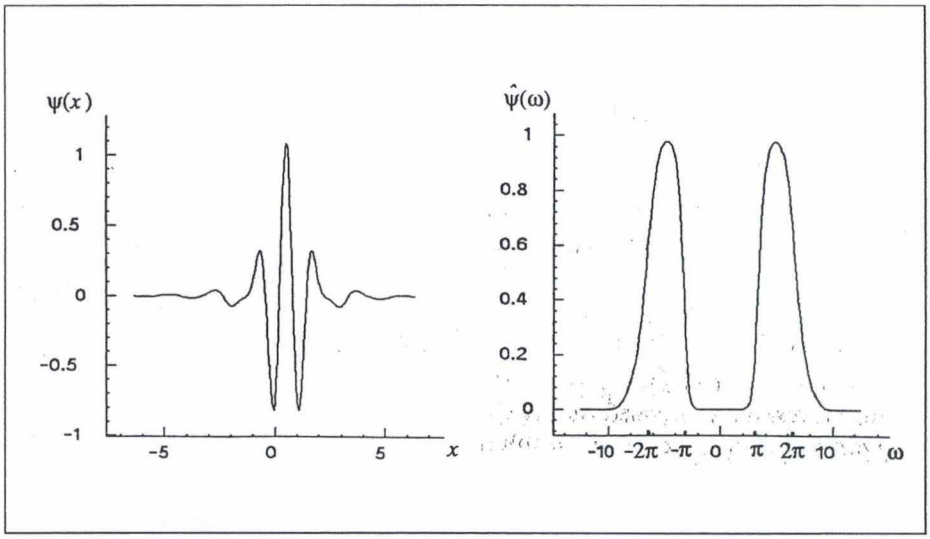


FIG. 2.5 – *Ondelette de Battle-Lemarié et sa Transformée de Fourier. On constate qu'il s'agit d'un filtre passe-bande.*

Comme la fonction d'échelle, cette Ondelette n'est pas continue. La figure 2.5 donne l'Ondelette de Battle-Lemarié, dont nous avons déjà vu $\phi(x)$ et $\mathbf{H}(\omega)$. On constate que cette ondelette se comporte comme un filtre passe-bande entre $[-2\pi, \pi]$ et $[\pi, 2\pi]$.

Nous sommes ici au coeur du problème : la construction d'Ondelettes utilisables en pratique. En effet, la méthode est la suivante : construire un filtre $\mathbf{H}(\omega)$ satisfaisant les conditions du théorème 2 permet de déduire $\phi(x)$, la fonction d'échelle. A partir de ces deux constructions, on peut, par le théorème 3, calculer $\psi(x)$. Les qualités de $\phi(x)$ et de $\psi(x)$ dépendent donc du choix de $\mathbf{H}(\omega)$. On parle ici de propriété de dérivabilité et de support compact. On trouvera dans [6, chapitre 5 et 6], l'étude détaillée de ces aspects de la question. Un résultat important est que $\forall n > 0$, $\exists \mathbf{H}(\omega)$ tel que $\psi(x)$, l'Ondelette associée soit à support compact et n fois continuemment différentiable. Remarquons que l'Ondelette de Battle-Lemarié n'est pas à support compact mais exponentiellement décroissante.

Définissons $\mathbf{P}_{\mathbb{O}_{2^j}}$ l'opérateur de projection sur $\mathbb{O}_{2^j}$ par la relation

$$\mathbf{P}_{\mathbb{O}_{2^j}} f(x) \;=\; 2^{-j} \sum_{x=-\infty}^{+\infty} \langle f(u),\; \psi_{2^j}(u - 2^{-j}n)\rangle . \psi_{2^j}(x - 2^{-j}n)$$

De la même manière que pour $\mathbf{A}^d_{2^j}$, définissons

$$\mathbf{D}_{2^j} f \;=\; (\langle f(u),\; \psi_{2^j}(u - 2^{-j}n)\rangle)_{n\in\mathbb{Z}}$$

que nous appellons l'ensemble discret des détails de $f(x)$ à la résolution 2^j. Cet ensemble contient la différence d'information entre $\mathbf{A}^d_{2^j}$ et $\mathbf{A}^d_{2^{j+1}}$. Nous pouvons dé-

velopper les produits scalaires pour prouver que chaque élément de $\mathbf{D}_{2^j} f$ est le produit de convolution entre $f(x)$ et $\psi_{2^j}(-x)$ évalué en $2^{-j}n$, soit :

$$\langle f(u),\ \psi_{2^j}(u - 2^{-j}n)\rangle \ = \ (f(u) * \psi_{2^j}(-u))(2^{-j}n)$$

comme nous l'avons déjà fait plus haut.

Ceci veut donc dire que les détails de $f(x)$ à l'aproximation 2^j peuvent être calculés par le passage de $f(x)$ dans le filtre passe-haut $\psi_{2^j}(-u)$ et l'échantillonage du résultat à la fréquence 2^j. Les détails, comme nous l'avons déjà remarqué, sont données par les hautes fréquences du signal.

A présent, nous pouvons reconstruire par induction l'approximation initiale $\mathbf{A}_1^d f$ par l'ensemble

$$(\mathbf{A}_{2^{-J}}^d f, (\mathbf{D}_{2^j})_{-J \leqslant j \leqslant -1})$$

On appellera cet ensemble la représentation en Ondelettes Orthogonales du signal $f(x)$. Elle est constituée de l'approximation du signal à la résolution 2^{-J} ajoutée des détails du signal aux résolutions allant de 2^{-J} à 2^{-1}. L'interprétation immédiate de ceci est que le signal a été découpé en bandes de fréquences indépendantes puisque les fonctions $\psi_{2^j}(x)$ sont des filtres passe-bande allant de $-2^j\pi$ à $-2^{j+1}\pi$ et de $2^j\pi$ à $2^{j+1}\pi$. L'indépendance des bandes de fréquences étant dûe à l'orthogonalité des fonctions $\psi_{2^j}(x)$.

2.3.2 Implémentation de la représentation en Ondelettes Orthogonales

A partir des considérations précédentes, nous pouvons à présent définir un algorithme pyramidal pour calculer la représentation en Ondelettes d'un signal. Pour ce faire, nous allons montrer qu'on peut obtenir les ensembles $\mathbf{D}_{2^j} f$ à partir des approximations déjà calculées précédemment.

$\forall n \in \mathbb{Z}$, la fonction $\psi_{2^j}(x - 2^{-j}n)$ appartient à l'ensemble $\mathbb{O}_{2^j} \subset \mathbb{V}_{2^{j+1}}$, donc, nous pouvons exprimer cette fonction sur une base orthogonale de $\mathbb{V}_{2^{j+1}}$, une base que nous avons déjà rencontrée.

$$\psi_{2^j}(x - 2^{-j}n) \ = \ 2^{-j-1} \sum_{k=-\infty}^{+\infty} \langle \psi_{2^j}(u - 2^{-j}n),\ \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle \ .\phi_{2^{j+1}}(x - 2^{-j-1}k)$$

Avec un changement de variable similaire à celui déjà employé pour $\mathbf{A}_{2^j}^d$, nous pouvons prouver que :

$$2^{-j-1}\langle \psi_{2^j}(u - 2^{-j}n),\ \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle \ = \langle \psi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle$$

Ceci nous donne donc que :

$$\psi_{2^j}(x - 2^{-j}n) \ = \ \sum_{k=-\infty}^{+\infty} \langle \psi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle \ .\phi_{2^{j+1}}(x - 2^{-j-1}k)$$

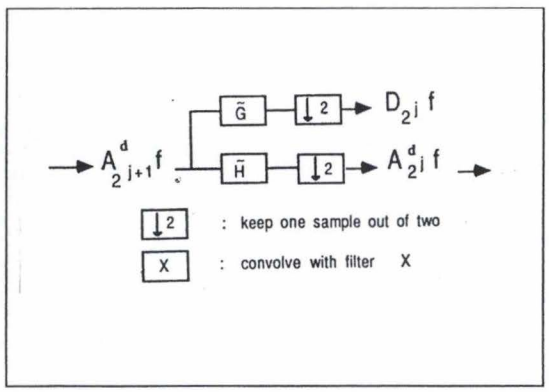


FIG. 2.6 – *Décomposition de* $\mathbf{A}^d_{2^{j+1}} f$ *en* $\mathbf{A}^d_{2^j} f$, *une approximation à résolution plus faible, et* $\mathbf{D}_{2^j} f$ *les détails à la résolution* 2^j. *L'algorithme est constitué par la répétition en cascade de cette opération pour* $-1 \geqslant j \geqslant -J$.

En prenant le produit scalaire des deux nombres avec $f(u)$, on obtient :

$$\langle f(u),\ \psi_{2^j}(u - 2^{-j}n)\rangle \ = \ \sum_{k=-\infty}^{+\infty} \langle \psi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle \ . \ \langle f(u),\ \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle$$

Ici, comme pour $\mathbf{H}$, nous posons que $\mathbf{G}$ est un filtre discret dont la réponse impulsionnelle est $g(n) \ = \langle \psi_{2^{-1}}(u),\ \phi(u - (k - 2n))\rangle$ et $\tilde{\mathbf{G}}$ sera, comme pour $\mathbf{H}$, le filtre symétrique dont la réponse impulsionnelle est inversée $\tilde{g}(n) \ = \ g(-n)$. On peut montrer que la fonction de transfert de ce filtre est la fenêtre $\mathbf{G}(\omega)$ définie dans l'énoncé du théorème 3 (la preuve se trouve résumée dans [25] et in extenso dans [24]).

Introduisons $g(n)$ dans l'équation précédente, on arrive à :

$$\langle f(u),\ \psi_{2^j}(u - 2^{-j}n)\rangle \ = \ \sum_{k=-\infty}^{+\infty} \tilde{g}(2n - k)\ \langle f(u),\ \phi_{2^{j+1}}(u - 2^{-j-1}k)\rangle$$

Cette équation équivaut à :

$$\langle f(u),\ \psi_{2^j}(u - 2^{-j}n)\rangle \ = (\tilde{g} * \mathbf{A}^d_{2^{j+1}} f)(2n) \ = \ (\mathbf{D}_{2^j} f)(n)$$

Donc, $\mathbf{D}_{2^j} f$ correspond à la convolution de $\mathbf{A}^d_{2^{j+1}} f$ avec le filtre $\tilde{\mathbf{G}}$ puis à l'échantillonnage du résultat avec une fréquence $\frac{1}{2}$.

Nous pouvons à présent écrire l'algorithme de façon naturelle. L'état initial est $\mathbf{A}^d_1 f$, pour une étape de l'algorithme, on décompose$\mathbf{A}^d_{2^{j+1}} f$ en $\mathbf{A}^d_{2^j} f$ par un passage dans $\tilde{\mathbf{H}}$ et échantillonage et en $\mathbf{D}_{2^j} f$ par passage dans $\tilde{\mathbf{G}}$ et échantillonage. On effectue ceci pour tout j compris entre -1 et $-J$. Cet algorithme est illustré par le schéma de la figure 2.6 .

On gèrera les problèmes dûs au nombre fini d'échantillons de $\mathbf{A}_{2^j}^d f$ par la même convention de symétrie par rapport à 0 et à N.

On peut prouver que $g(n) = (-1)^{1-n} h(1-n)$ (cfr [25]). Les filtres $\mathbf{H}$ et $\mathbf{G}$ sont appelés filtres miroirs. $\mathbf{H}$ est un filtre passe-bas et $\mathbf{G}$ est un filtre passe-haut.

Remarquons que, contrairement à certains algorithme pyramidaux, notre résultat :

$$(\mathbf{A}_{2^{-J}}^d f,\ (\mathbf{D}_{2^j} f)_{-J \leqslant j \leqslant -1})$$

comporte le même nombre d'échantillons que $\mathbf{A}_1^d f$ puisqu'à chaque étape nous ne prenons qu'un échantillon sur deux.

2.3.3 Reconstruction du signal depuis sa représentation en Ondelettes Orthogonales

La Transformée en Ondelettes Continue étant inversible, nous nous attendons à ce que notre transformation le soit aussi : c'est ce que ce paragraphe va montrer, en donnant l'algorithme nécessaire pour y parvenir. Nous savons que $\mathbb{O}_{2^j}$ est le complément orthogonal de $\mathbb{V}_{2^j}$ dans $\mathbb{V}_{2^{j+1}}$. Donc, $(\sqrt{2^{-j}}\phi_{2^j}(u - 2^{-j}n),\ \sqrt{2^{-j}}\psi_{2^j}(u - 2^{-j}n))_{n\in\mathbb{Z}}$ est une base orthonormale de $\mathbb{V}_{2^{j+1}}$.

On peut donc écrire $\phi_{2^{j+1}}(x - 2^{-j-1}n) \in \mathbb{V}_{2^{j+1}}$ sous la forme :

$$\begin{aligned}
&\phi_{2^{j+1}}(x - 2^{-j-1}n) \ = \\
&\qquad 2^{-j} \textstyle\sum_{k=-\infty}^{+\infty} \langle \phi_{2^j}(u - 2^{-j}k),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ .\ \phi_{2^j}(x - 2^{-j}k) \\
&\qquad + 2^{-j} \textstyle\sum_{k=-\infty}^{+\infty} \langle \psi_{2^j}(u - 2^{-j}k),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ .\ \psi_{2^j}(x - 2^{-j}k)
\end{aligned}$$

Comme toujours, prenons le produit scalaire des deux membres avec $f(x)$:

$$\begin{aligned}
&\langle f(u),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ = \\
&\qquad 2^{-j} \textstyle\sum_{k=-\infty}^{+\infty} \langle \phi_{2^j}(u - 2^{-j}k),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ .\ \langle f(u),\ \phi_{2^j}(u - 2^{-j}k)\rangle + \\
&\qquad 2^{-j} \textstyle\sum_{k=-\infty}^{+\infty} \langle \psi_{2^j}(u - 2^{-j}k),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ .\langle f(u),\ \psi_{2^j}(u - 2^{-j}k)\rangle
\end{aligned}$$

En utilisant les résultats obtenus précedemment, pour $\mathbf{H}$ et $\mathbf{G}$ et en les insérant dans cette formule, on obtient :

$$\begin{aligned}
&\langle f(u),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ = \\
&\qquad 2 \textstyle\sum_{k=-\infty}^{+\infty} h(n-2k)\langle f(u), \phi_{2^j}(u - 2^{-j}k)\rangle + \\
&\qquad 2 \textstyle\sum_{k=-\infty}^{+\infty} g(n-2k)\langle f(u),\ \psi_{2^j}(u - 2^{-j}k)\rangle
\end{aligned}$$

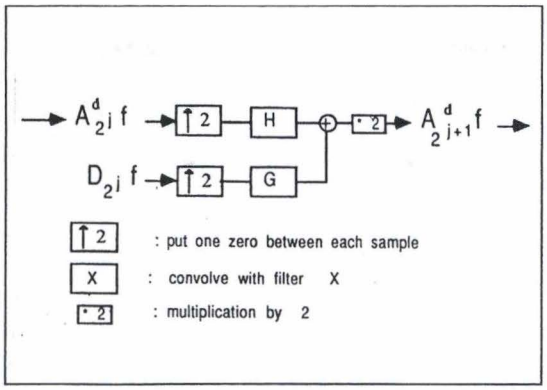


FIG. 2.7 – *Reconstruction de* $\mathbf{A}^d_{2^{j+1}} f$ *à partir de* $\mathbf{A}^d_{2^j}$ *et* $\mathbf{D}_{2^j}$. *Le signal de départ peut donc être reconstruit en répétant cette opération en cascade depuis* $-J$ *jusque* -1.

Soit

$$\langle f(u),\ \phi_{2^{j+1}}(u - 2^{-j-1}n)\rangle \ = \ 2 \sum_{k=-\infty}^{+\infty} h(n-2k)(\mathbf{A}^d_{2^j} f)(k) \ + \ 2 \sum_{k=-\infty}^{+\infty} g(n-2k)(\mathbf{D}_{2^j} f)(k)$$

Ce que l'on interprétera par le fait que $\mathbf{A}^d_{2^{j+1}} f$ peut être reconstruit par insertion d'un zéro entre les échantillons $\mathbf{A}^d_{2^j} f$ et $\mathbf{D}_{2^j} f$ puis par convolution des signaux résultants avec $\mathbf{H}$ et $\mathbf{G}$ respectivement. On termine par une sommation et une multiplication par 2. Ce procédé est illustré à la figure 2.7 . En répétant cette procédure depuis $-J$ jusque 0, on reconstruit $\mathbf{A}^d_1 f$.

Je suis personnellement soufflé de la simplicité et de l'élégance de cette théorie.

2.4 Extension à deux dimensions de la Représentation en Ondelettes Orthogonales

Dans le cadre du traitement d'images, il convient d'élargir la théorie au cas bi-dimmensionnel. En réalité, notre transformation peut se généraliser aisément à n'importe quel nombre de dimensions comme le montre Meyer dans [27]. Dans la suite de la théorie, Mallat donne une élégante manière de procéder pour le cas des images à deux dimensions. Suivons le donc à nouveau.

Les fonctions que nous étudierons sont cette fois des fonctions de $\mathbf{L}^2(\mathbb{R}^2)$. De la même manière que pour $\mathbf{L}^2(\mathbb{R})$, on définit une Approximation Multirésolution de $\mathbf{L}^2(\mathbb{R}^2)$ comme une séquence de sous-espaces de $\mathbf{L}^2(\mathbb{R}^2)$ satisfaisant, à deux dimensions, l'équivalent des propriétés vues au point 4.1. Soit $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ une Approximation Multirésolution de $\mathbf{L}^2(\mathbb{R}^2)$. L'approximation de $f(x,y) \in \mathbf{L}^2(\mathbb{R}^2)$ à la résolution 2^j est simplement la projection orthogonale de $f(x,y)$ sur $\mathbb{V}_{2^j}$. Le théorème 1 est toujours valide à deux dimensions et nous pouvons donc en tirer qu'il existe une fonction

$\Phi(x,y)$ dont les dilatations et les translations forment une base orthonormale de $\mathbb{V}_{2^j}, \forall j \in \mathbb{Z}$.

Pour exprimer ces bases, définissons de manière naturelle

$$\Phi_{2^j}(x,y) \;=\; 2^j \Phi(2^j x, 2^j y)$$

On a donc que :

$$(2^{-j}\Phi_{2^j}(x - 2^{-j}n, y - 2^{-j}m))_{(n,m)\in\mathbb{Z}^2}$$

forme une base orthonormale de $\mathbb{V}_{2^j}$. Tout comme à une dimension, la fonction Φ est unique pour une Approximation Multirésolution donnée. Meyer a décrit dans [26] un type d'Approximation Multirésolution particulier pour $\mathbf{L}^2(\mathbb{R}^2)$ qui est particulièrement adapté à l'étude des images. Dans son étude, chaque $\mathbb{V}_{2^j}$ peut s'écrire sous la forme $\mathbb{V}_{2^j} : \mathbb{V}^1_{2^j} \oplus \mathbb{V}^1_{2^j}$ où $\oplus$ dénote le produit Tensoriel et $\mathbb{V}^1_{2^j}$ est un sous-espace de $\mathbf{L}^2(\mathbb{R}^2)$. Dans ces conditions, il est clair que $\mathbb{V}_{2^j}$ n'est une Approximation Multirésolution que si $\mathbb{V}^1_{2^j}$ l'est aussi. De même, on peut déduire que :

$$\Phi(x,y) \;=\; \phi(x)\phi(y)$$

où $\phi(x)$ est une fonction d'échelle d'une Approximation Multirésolution $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ de $\mathbf{L}^2(\mathbb{R})$. Avec ce choix particulier d'Approximation Multirésolution *séparable*, les directions horizontales et verticales sont choisies préférentiellement. Ce choix est arbitraire mais justifié pour la plupart des images générées par l'être humain puisque ces directions sont en général privilégiées.

On peut bien sûr réécrire la base orthonormale de $\mathbb{V}_{2^j}$ sous la forme

$$(2^{-j}\Phi_{2^j}(x - 2^{-j}n, y - 2^{-j}m))_{(n,m)\in\mathbb{Z}^2} \;=\; (2^{-j}\phi_{2^j}(x - 2^{-j}n)\phi_{2^j}(y - 2^{-j}m))_{(n,m)\in\mathbb{Z}^2}$$

On caractérisera donc l'approximation du signal $f(x,y)$ à la résolution 2^j par

$$\mathbf{A}^d_{2^j} f \;=\; (\langle f(x,y),\; \phi_{2^j}(x - 2^{-j}n)\phi_{2^j}(y - 2^{-j}m)\rangle)_{(n,m)\in\mathbb{Z}^2}$$

Comme à une dimension, $\mathbf{A}^d_1 f$ sera le signal (l'image) de départ, prise par l'appareil de saisie. Supposons que cette image contienne N pixels. Il paraît clair que l'approximation $\mathbf{A}^d_{2^j} f$ contient $2^j N$ pixels. Considérons que les problèmes de bordure seront résolus par symétrie par rapport aux bords de l'image de la même manière qu'à une dimension.

Pour exprimer les détails du signal à la résolution 2^j nous devons, à la suite de Mallat, étendre le théorème 3. Soit $\mathbb{O}_{2^j}$ le complément orthogonal de $\mathbb{V}_{2^j}$ dans $\mathbb{V}_{2^{j+1}}$.

Théorème 4

Soit $(\mathbb{V}_{2^j})_{j\in\mathbb{Z}}$ une Approche Multirésolution séparable de $\mathbf{L}^2(\mathbb{R}^2)$.

Soit $\Phi(x,y) = \phi(x)\phi(y)$ la fonction d'échelle associée. Alors

$$\begin{aligned}
\Psi^1(x,y) &= \phi(x)\psi(y) \\
\Psi^2(x,y) &= \psi(x)\phi(y) \\
\Psi^3(x,y) &= \psi(x)\psi(y)
\end{aligned}$$

sont telles que

$$\begin{pmatrix} 2^{-j}\Psi^1_{2^j}(x-2^{-j}n, y-2^{-j}m), \\ 2^{-j}\Psi^2_{2^j}(x-2^{-j}n, y-2^{-j}m) \\ 2^{-j}\Psi^3_{2^j}(x-2^{-j}n, y-2^{-j}m) \end{pmatrix}_{(n,m)\in\mathbb{Z}^2}$$

est une base orthonormale de $\mathbb{O}_{2^j}$ et

$$\begin{pmatrix} 2^{-j}\Psi^1_{2^j}(x-2^{-j}n, y-2^{-j}m), \\ 2^{-j}\Psi^2_{2^j}(x-2^{-j}n, y-2^{-j}m) \\ 2^{-j}\Psi^3_{2^j}(x-2^{-j}n, y-2^{-j}m) \end{pmatrix}_{(j,n,m)\in\mathbb{Z}^3}$$

est une base orthonormale de $\mathbf{L}^2(\mathbb{R}^2)$

■

La preuve de ce théorème se trouve dans [24].

Continuons simplement à étendre le cas uni-dimensionnel à partir de ceci. Les détails du signal à la résolution 2^j sont donnés par la projection orthogonale de $f(x,y)$ sur $\mathbb{O}_{2^j}$. Cette fois, si nous exprimons cette projection à l'aide de la base que nous donne le théorème 4, nous obtenons trois ensembles de coefficients donnés par le produit scalaire de $f(x,y)$ avec chacune des trois Ondelettes. Définissons

$$\begin{aligned} \mathbf{D}^1_{2^j} f &= (\langle f(x,y),\ \Psi^1_{2^j}(x-2^{-j}n, y-2^{-j}m)\rangle)_{(n,m)\in\mathbb{Z}^2} \\ \mathbf{D}^2_{2^j} f &= (\langle f(x,y),\ \Psi^2_{2^j}(x-2^{-j}n, y-2^{-j}m)\rangle)_{(n,m)\in\mathbb{Z}^2} \\ \mathbf{D}^3_{2^j} f &= (\langle f(x,y),\ \Psi^3_{2^j}(x-2^{-j}n, y-2^{-j}m)\rangle)_{(n,m)\in\mathbb{Z}^2} \end{aligned}$$

Comme nous l'avons déjà fait deux fois à une dimension, nous pouvons montrer que chacun des produits scalaires est en fait une convolution. Nous réécrirons donc finalement $\mathbf{A}^d_{2^j} f$, $\mathbf{D}^1_{2^j} f$, $\mathbf{D}^2_{2^j} f$ et $\mathbf{D}^3_{2^j} f$ sous la forme

$$\begin{aligned} \mathbf{A}^d_{2^j} f &= ((f(x,y) * \phi_{2^j}(-x)\phi_{2^j}(-y))(2^{-j}n, 2^{-j}m))_{(n,m)\in\mathbb{Z}^2} \\ \mathbf{D}^1_{2^j} f &= ((f(x,y) * \phi_{2^j}(-x)\psi_{2^j}(-y))(2^{-j}n, 2^{-j}m))_{(n,m)\in\mathbb{Z}^2} \\ \mathbf{D}^2_{2^j} f &= ((f(x,y) * \psi_{2^j}(-x)\phi_{2^j}(-y))(2^{-j}n, 2^{-j}m))_{(n,m)\in\mathbb{Z}^2} \\ \mathbf{D}^3_{2^j} f &= ((f(x,y) * \psi_{2^j}(-x)\psi_{2^j}(-y))(2^{-j}n, 2^{-j}m))_{(n,m)\in\mathbb{Z}^2} \end{aligned}$$

Nous arrivons donc enfin au coeur de la transformation, ces quatre relations nous donnent la manière d'obtenir ces coefficients. Ici, la décomposition en Ondelettes peut être interprétée comme une décomposition du signal en ensembles de fréquences indépendantes et orientées spacialement. En effet, si nous supposons que $\phi(x)$ et $\psi(x)$ sont des filtres parfaits passe-bas et passe-bande, la figure 2.8 (a) nous montre les domaines de fréquences pour l'image $\mathbf{A}^d_{2^{j+1}} f$, décomposée en $\mathbf{A}^d_{2^j} f$, $\mathbf{D}^1_{2^j} f$, $\mathbf{D}^2_{2^j} f$ et $\mathbf{D}^3_{2^j} f$. $\mathbf{A}^d_{2^j} f$ reprend les basses fréquences dans les deux directions, $\mathbf{D}^1_{2^j} f$ reprend les hautes fréquences verticales (correspondants aux bords horizontaux), $\mathbf{D}^2_{2^j} f$, les hautes fréquences horizontales (correspondants aux bords verticaux), et $\mathbf{D}^3_{2^j} f$, les hautes fréquences dans les deux sens (correspondants aux coins). La figure 2.9 illustre

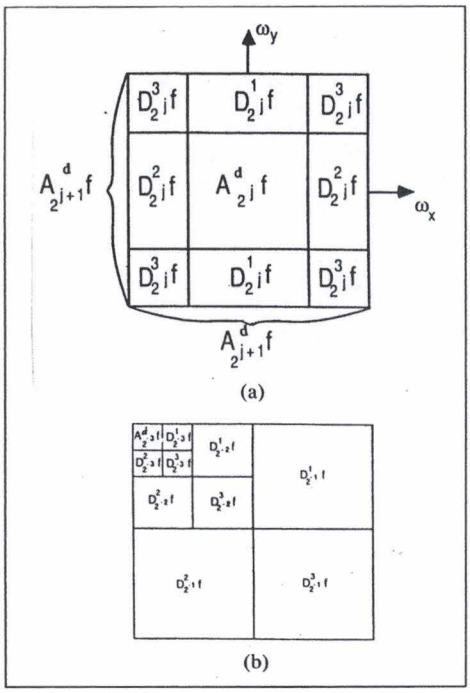


FIG. 2.8 – *(a)Décomposition du domaine de fréquences d'une image. (b)Arrangement des bandes de fréquences pour obtenir unr Représentation en Ondelettes.*

ceci par la décomposition d'un rectangle clair sur fond noir. Les bandes de fréquences sont arrangées comme sur la figure 2.8 (b).

On a donc que $\forall J > 0$, $\mathbf{A}_1^d f$ est représentée par $3J + 1$ images :

$$(\mathbf{A}_{2^{-J}}^d f,\ (\mathbf{D}_{2^j}^1 f)_{-J \leqslant j \leqslant -1},\ (\mathbf{D}_{2^j}^2 f)_{-J \leqslant j \leqslant -1},\ (\mathbf{D}_{2^j}^3 f)_{-J \leqslant j \leqslant -1})$$

Cette représentation n'augmente pas le nombre de pixels puisque si $\mathbf{A}_1^d f$ possède N pixels, $\mathbf{A}_{2^j}^d f$, $\mathbf{D}_{2^j}^1 f$, $\mathbf{D}_{2^j}^2 f$ et $\mathbf{D}_{2^j}^3 f$ possèdent $2^j N$ pixels.

Nous pouvons maintenant exprimer les algorithmes de décomposition et de reconstruction à deux dimensions, ceux-ci sont très similaires à ceux déduit à une dimension. On peut, selon Mallat, voir cette transformation bi-dimensionnelle comme une transformation unidimensionnelle selon chaque axe. A chaque étape, $\mathbf{A}_{2^{j+1}}^d f$ est décomposé en $\mathbf{A}_{2^j}^d f$, $\mathbf{D}_{2^j}^1 f$, $\mathbf{D}_{2^j}^2 f$ et $\mathbf{D}_{2^j}^3 f$, comme illustré à la figure 2.10 . Il s'agit simplement de convoluer les lignes avec les filtres $\tilde{\mathbf{G}}$ et $\tilde{\mathbf{H}}$, de ne retenir qu'un élément sur deux puis de convoluer les colonnes des résultats avec les mêmes filtres pour obtenir les quatre sous-images. $\tilde{\mathbf{G}}$ et $\tilde{\mathbf{H}}$ sont les mêmes que ceux utilisés plus haut. L'algorithme de reconstruction se fait de manière similaire et est illustré à la figure 2.11 . Pour les deux algorithmes, on passe de $-J$ à 0 et de 0 à $-J$ respectivement suivant la structure pyramidale habituelle.

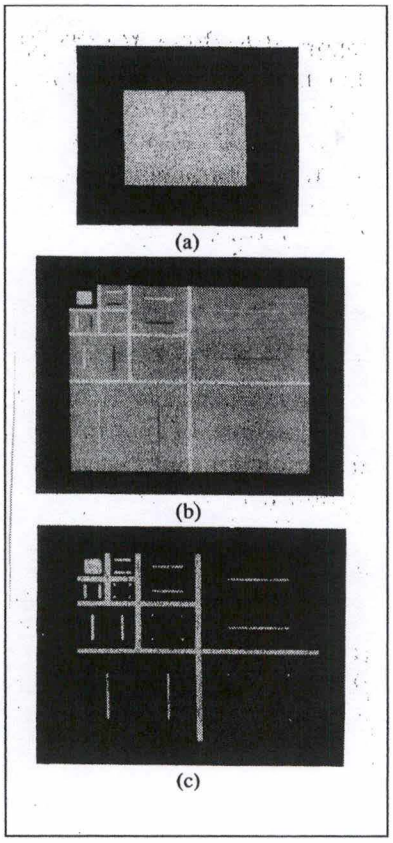


FIG. 2.9 – *(a) Image originale. (b) Représentation en Ondelettes de cette image. Les pixels noirs correspondent aux coefficients nuls et les blancs aux positifs. (c) Même chose mais en valeur absolue. On constate que l'amplitude des coefficients est grande le long des bords dans chaque orientation.*

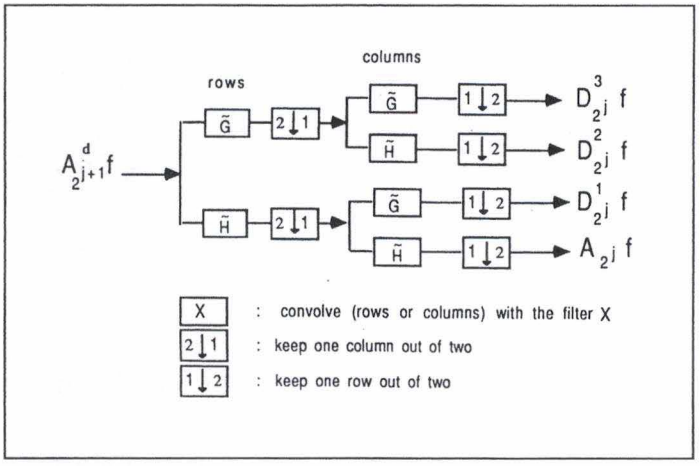


FIG. 2.10 – *Une étape de l'algorithme de décomposition.*

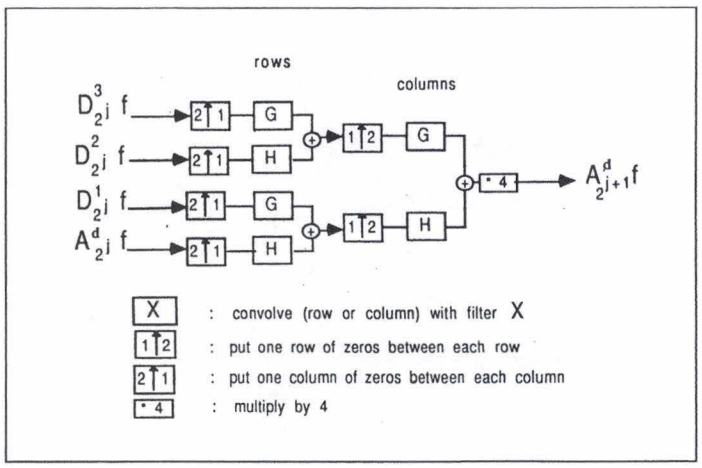


FIG. 2.11 – *Une étape de l'algorithme de reconstruction.*

2.5 Application au codage compact des images

Le codage d'images via les algorithmes que nous venons de décrire a déjà été intensivement étudié [2, 41, 29, 28, 40, 34, 39, 37, 50, 42, 38]. Les différents schémas de compression utilisent peu ou prou la même transformation de base et se servent avec plus ou moins de bonheur des propriétés des coefficients ainsi calculés pour gagner de l'espace de stockage. Le principe est toujours, dans un premier temps, d'introduire des erreurs dans la représentation en Ondelettes. Lors de la reconstruction, si on a bien fait son travail, les erreurs ne devraient pas produire d'anomalies visibles pour l'être humain. Dans ses études fondatrices [25, 24], Mallat nous offrait déjà la théorie nécessaire à toutes ces techniques, théorie qui a été largement raffinée par la suite. Dans le cadre de ce mémoire, je pense que nous pouvons nous en tenir à ces bases.

Nous étudions donc les erreurs introduites par la représentation d'une image par Transformée en Ondelettes Discrète. Soit $\mathbf{A}^d_{2^{-J}} f$, $(\mathbf{D}^1_{2^j} f)_{-J \leqslant j \leqslant -1}$, $(\mathbf{D}^2_{2^j} f)_{-J \leqslant j \leqslant -1}$, $(\mathbf{D}^3_{2^j} f)_{-J \leqslant j \leqslant -1}$ la représentation en Ondelettes d'une image, notons $(\varepsilon_{-J}$, $(\varepsilon^1_j)_{-J \leqslant j \leqslant -1}$, $(\varepsilon^2_j)_{-J \leqslant j \leqslant -1}$, $(\varepsilon^3_j)_{-J \leqslant j \leqslant -1})$ les erreurs (écarts quadratiques moyens) introduites en codant chaque composante de l'image. Soit ε_0 l'erreur de l'image reconstruite à partir de la transformation par rapport à l'image originale. On peut montrer [24] que :

$$\varepsilon_0 = 2^{2J} \varepsilon_{-J} + \sum_{k=1}^{3} \sum_{j=J}^{-1} 2^{-2j} \varepsilon^k_j$$

De nombreuses études ont montré que la sensibilité de l'oeil humain à la distortion n'est pas seulement fonction de l'erreur ε_0 mais aussi de la distribution de cette erreur à travers l'image, des fréquences spatiales modifiées et de l'orientation des erreurs [13, 14] (dans ce dernier cas, on montre que la sensibilité est minimale pour des stimuli orientés à 45 degrés sur l'horizontale). Le codage de la Représentation en Ondelettes est orienté, il est possible de l'adapter à l'oeil humain.

En traitement du signal, un grand problème est celui de décrire le sous-ensemble de $\mathbf{L}^2(\mathbb{R}^2)$ que forment les images naturelles [16]. En effet, l'étude expérimentale de ces images montre qu'elles possèdent des propriétés communes et de nombreuses études tendent à définir celles-ci. Mallat a effectué une étude sur plusieurs images naturelles et montre que les coefficients des représentations en Ondelettes de ces images se répartissent sur des histogrammes particuliers, centrés autour de zéro, quelque soit la résolution ou l'orientation que l'on choisisse. Cette constatation est importante car la Représentation en Ondelettes est une décomposition des pixels de l'image sur des bases orthonormales indépendantes et donc, les coefficients obtenus ne sont pas corrélés. On peut donc se servir de la Représentation en Ondelettes pour caractériser les images naturelles. Les résultats de Mallat, qui ont été largement repris dans la littérature, sont que les histogrammes des coefficients des Représentations en Ondelettes d'images naturelles peuvent se modéliser par la famille d'histogrammes

$$h(u) \;=\; \mathbf{K}\, e^{-(\frac{|u|}{\alpha})^{\beta}}$$

Dans cette formule, α modélise la variance et β règle la décroissance du pic. La constante $\mathbf{K}$ doit être ajustée pour obtenir $\int_{-\infty}^{+\infty} h(u)\mathrm{d}u \;=\; N$ où N est le nombre total de pixels de la sous-partie de représentation en Ondelettes étudiée. La figure 2.12 montre un histogramme typique d'image et son approximation via $h(u)$ (la méthode de détermination de $\mathbf{k}, \alpha$ et β à partir de l'histogramme "expérimental" peut être trouvé dans [24]).

Il est clair que la connaissance d'une forme mathématique simple pour les histogrammes des Représentations en Ondelettes est, comme nous allons le voir plus loin, un avantage majeur dans le codage d'images.

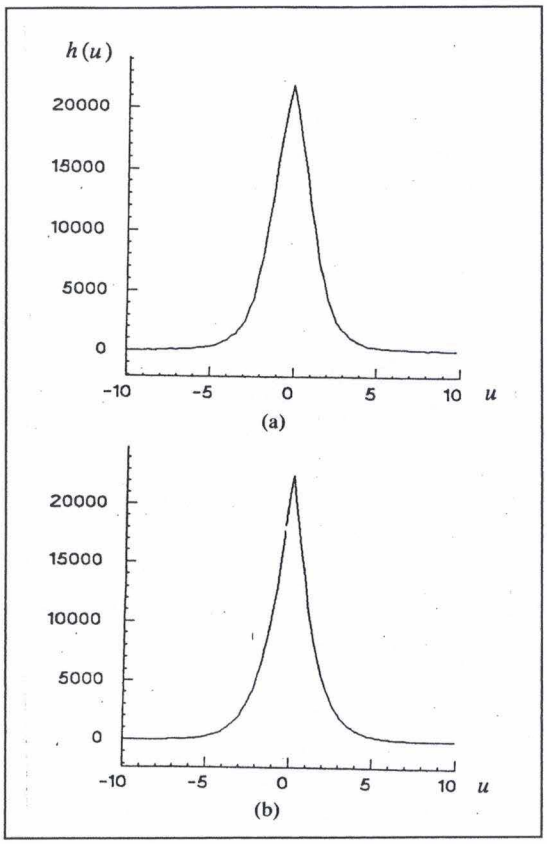


FIG. 2.12 – *Histogramme typique d'image et de son approximation.*

Chapitre 3

Application : la compression d'images

L'analyse multirésolution nous a fourni un outil formidable pour le traitement d'images ; il est temps à présent de voir comment on s'en sert effectivement pour la compression.

Dans ce chapitre, je vais exposer les grands principes de la compression d'images avec la Transformée en Ondelettes. Nous effectuerons une comparaison entre cette méthode et celle utilisée dans le standard JPEG. Ensuite, je présenterai de manière détaillée les programmes que j'ai réalisé et j'exposerai les résultats obtenus. Enfin, comme je l'ai mentionné plus haut, les dernières étapes de codage n'ont pas été reprogrammées mais je les exposerai rapidement ainsi que les résultats obtenus par leurs auteurs. J'insisterai plus particulièrement sur la méthode EZW qui est la plus connue.

3.1 Compression d'images avec la Transformée en Ondelettes

La compression d'images a été l'une des premières applications de la Transformée en Ondelettes Discrète. Elle a été développée par nombre d'auteurs ([41, 29, 28, 40, 34, 39, 37, 50, 42, 38, 36]) et connait aujourd'hui un grand succès dans de nombreux domaines. Pour l'appréhender, nous allons commencer par voir ce qu'il en est pour le standard JPEG, tout au moins dans ses grands principes.

3.1.1 Compression JPEG

Dans la compression JPEG, l'image, la matrice des pixels est tout d'abord découpée en matrices de 8x8 pixels M_{ij} (ou plus rarement en matrice 16x16).

Chaque bloc va ensuite subir une Transformée en Cosinus Discrète à deux dimensions. Cette transformation, proche de la Transformée de Fourier, est tout à fait

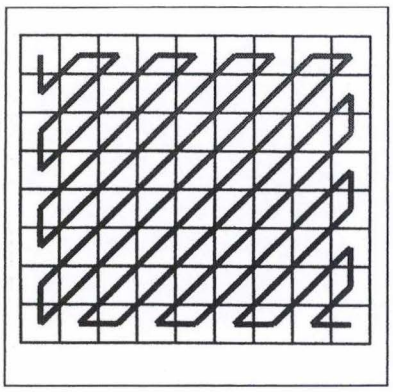


FIG. 3.1 – *Codage de M''_{ij} en zig-zag.*

inversible (aux conditions habituelles des signaux discrets). Elle produit une matrice (M'_{ij}) de coefficients réels très dispersés, dont beaucoup de coefficients sont proches de zéro.

Grâce à une matrice de quantification Q_{ij}, on va mettre à zéro les plus petits coefficients de la matrice. La formule est :

$$Q_{ij}.\mathbf{Arrondi}(M'_{ij}/Q_{ij})$$

La matrice obtenue peut subir une Transformée en Cosinus inverse. Le résultat sera proche de la matrice de départ mais différent de celui-ci : on a une perte de détails. Le codage effectif de la matrice M''_{ij} se fait en profitant de sa structure particulière, induite par la Transformée en Cosinus Discrète et l'utilisation de matrices Q_{ij} spécifiques. En effet, les coefficients non-nuls se trouvent tous dans les indices faibles en i et j, de plus, ils sont proches les uns des autres. On code donc la valeur du plus grand d'entre eux, M''_{00}. Les suivants seront codés par leur différence par rapport à ce dernier.

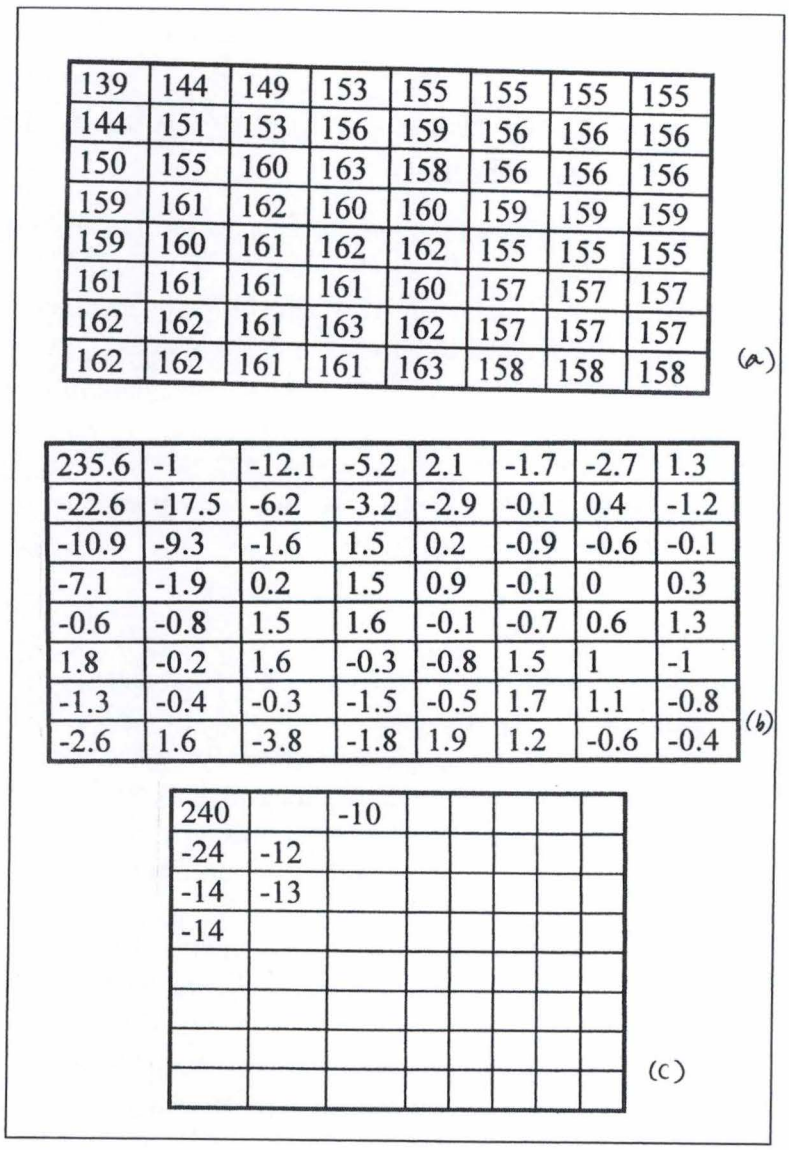
L'ordre dans lequel on les code est lui aussi fonction de la structure de la matrice M''_{ij}. En effet, les éléments non-nuls sont disposés suivant des "diagonales" $M''_{k,0} - M''_{0,k}$. On code donc les éléments suivant une séquence en zig-zag, comme indiqué sur la figure 3.1 , tirée de [17]. On arrête le codage lorsqu'on a codé tous les éléments non-nuls, on insère alors un symbole de fin de bloc (EOB).

Enfin, la liste obtenue sera effectivement encodée grâce à l'algorithme entropique de Huffmann.

La figure 3.2 tirée de [17] montre l'évolution d'une matrice 8x8 à travers le traitement.

Il reste finalement à insérer, devant les données, un header contenant toutes les informations utiles au décodage.

L'algorithme JPEG est très efficace et c'est ce qui en fait un standard aujourd'hui. Cependant, il souffre d'un défaut pour les forts taux de compression que l'on appelle blocking artifacts. Ce défaut est dû à l'apparition de fortes distorsions aux



139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

(a)

235.6	-1	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1	-1
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

(b)

240		-10					
-24	-12						
-14	-13						
-14							

(c)

FIG. 3.2 – *Evolution d'une matrice 8x8 à travers le codage JPEG (a) matrice originale (b) M'_{ij} (c) M''_{ij}.*

frontières des blocs 16x16 ou 8x8. Or, l'utilisation des blocs est absolument nécessaire à l'algorithme JPEG. En effet, la Transformée en Cosinus, comme la Transformée de Fourier, est non-locale. La décomposition en sous-matrices est nécessaire pour que les pertes de qualités restent localisées et ne se répercutent pas sur l'ensemble de l'image. Nous allons ici pouvoir utiliser l'adaptation de la Transformée en Ondelettes Discrète aux variations locales pour outrepasser ces limitations.

3.1.2 Principes de la compression d'images via la Transformée en Ondelettes Discrète

La compression via la Transformée en Ondelettes Discrète va suivre, en gros, les grandes étapes de la compression JPEG. Cependant, il y est fait usage d'une propriété cruciale des Ondelettes : leur localisation en temps et en fréquence.

En effet, cette fois, nous utilisons la Transformée en Ondelettes Discrète et donc, nous pouvons envisager de traiter l'entièreté de l'image en une seule fois. Et c'est ce qui constitue la première étape du codage.

Une fois cette étape réalisée, nous n'allons pas modifier la répartition des coefficients c_{ij} dont nous avons un modèle fidèle et prometteur. La mise à zéro va se faire via une simple comparaison par rapport à un seuil s.

$$|c_{ij}| < s \Rightarrow c'_{ij} = 0$$

Ce type de traitement sera très efficace au vu de l'histogramme de répartition des coefficients de la représentation en Ondelettes (cfr section 4.5).

A première vue, la matrice obtenue semble ne présenter aucune structure. Cependant, on peut constater que, même avec des pourcentages énormes de coefficients mis à zéro (jusqu'à 95%), l'image reconstruite reste d'une grande fidélité à l'originale. Nous le verrons tout à l'heure avec les résultats expérimentaux obtenus dans le cadre de ce travail.

Le codage en zig-zag par différence par rapport à l'élément $(0, 0)$ est évidemment très spécifique à la norme JPEG et ne peut être utilisé ici. Par contre, nous pouvons utiliser les propriétés spécifiques de la Représentation en Ondelettes. De nombreuses études ([41, 40, 34, 39, 37, 50, 42, 38, 36]) présentent des algorithmes de codage très efficaces dans cette optique. Le premier d'entre eux [41], proposé par Jérôme Shapiro en 1993, est réellement fondamental car il a orienté toutes les recherches à ce jour. Cet algorithme, nommé EZW, sera présenté succintement à la section 5.3.

L'encryptage final de la suite de symboles obtenue se fait souvent par codage arithmétique mais également par des techniques spécifiques.

Ainsi la compression d'une image via la Transformée en Ondelettes Discrète se fait en quatre étapes :

1. La Transformée en Ondelettes Discrète
2. La mise à zéro des coefficients non significatifs

3. La construction d'une suite de symboles via un algorithme spécifique (par exemple EZW)
4. L'encryptage via codage arithmétique (ou autre).

3.2 Programmes réalisés et résultats

Les programmes que j'ai réalisés dans le cadre de ce travail avaient pour but de vérifier les potentialités de la Transformée en Ondelettes pour la compression d'images. Mon objectif était de pouvoir appliquer la Transformée en Ondelettes Discrète à une image, de filtrer les coefficients de la matrice obtenue et de reconstruire une image pour la comparer avec l'original. Cet objectif a été pleinement atteint.

Dans un premier temps, j'ai été amené à programmer en JAVA. C'était l'occasion d'utiliser les spécificités d'un langage orienté objet dont je maîtrise bien les concepts principaux. Ainsi, j'ai développé un programme qui se présente sous la forme d'une toolkit à destination du programmeur. En effet, à ma connaissance, aucun package JAVA ne contient d'outils relatifs à la Transformée en Ondelettes. J'ai développé un package WaveletTools qui offre un ensemble de fonctions permettant de manipuler des fichiers d'images au format bitmap Windows, d'appliquer la Transformée en Ondelettes Discrète dans les deux sens et de filtrer les coefficients pour mettre à zéro ceux inférieurs à un certain seuil. J'ai fait en sorte que ces outils soient entièrement configurables et flexibles, de façon à pouvoir servir dans de nombreuses circonstances. Chaque fonction présente de nombreuses possibilités d'utilisation qui sont configurables par le programmeur. Le code est, en outre, amplement commenté.

Ainsi, le package WaveletTools contient sept classes :

1. BitmapHandler
2. WTTransformer
3. DataCompressor
4. ImageCompressor
5. DataWavelet
6. WaveHandler
7. WaveCompressor

Les cinq premières sont consacrées à la compression d'images, les deux dernières à la compression de sons (cfr annexe D).

Dans un second temps, j'ai pu, pour augmenter les performances du programme, en reprogrammer certaines parties en FORTRAN90. Afin d'être cohérent, j'ai complété ces parties pour que l'ensemble puisse former une toolkit FORTRAN90. En effet, ce langage a été conçu pour être optimisé et est donc particulièrement adapté pour les calculs liés à la Transformée en Ondelettes Discrète. De plus, les capacités orientés objet du FORTRAN90 m'ont permis de programmer de manière élégante. Au final, la toolkit en FORTRAN présente presque toutes les fonctionnalités de celle

en JAVA mais ses performances, en terme de temps d'exécution, sont nettement supérieures.

Ajoutons enfin que j'ai tenu à n'utiliser que des programmes non-commerciaux tout au long de la réalisation de ce travail. Le JDK 1.2 de SUN MicroSystem pour compiler le JAVA, le compilateur VASTF90 pour le FORTRAN 90 et le tout tournant sur le système LINUX (RedHat 6.1). Les images étaient visionnées avec le viewer GNU GQview.

Dans cette section, je présenterai le package WaveTools avec toutes ses fonctionnalités dans le détail. La toolkit en FORTRAN90 ne fera pas l'objet d'une telle présentation puisque, peu ou prou, elle présente les mêmes fonctionnalités. J'en montrerai tout de même le schéma.

Le code commenté de ces programmes se trouve en annexe E.

3.2.1 Le packageWaveletTools

La figure 3.3 montre le schéma de classe de ce package.

Chaque classe fonctionne à la manière d'un objet autonome : la plupart de ses attributs sont privés et des méthodes publiques permettent d'effectuer les opérations permises sur ceux-ci. Les fonctions de ces classes agissent sur leurs attributs. Pour obtenir le résultat d'une fonction, il faut mettre les attributs aux bonnes valeurs, appeller la fonction voulue et récupérer les valeurs des attributs modifiés. Cette démarche, si elle est un peu lourde, permet une programmation très structurée en séparant les compétences de chaque classe et en contrôlant strictement les accès aux données internes.

Mon propos n'est pas ici de produire une documentation de style *javadoc*. Mon code source est disponible et suffisamment commenté que pour être compréhensible. Je vais donc faire une présentation des principales fonctionnalités et possibilités offertes par les différentes classes. Je pense toutefois qu'une bonne compréhension du fonctionnement du programme ne peut être atteinte que si l'on dispose du code devant soi en lisant ces lignes.

1. `BitmapHandler` : la classe de gestion des images Bitmap

   Le format bitmap (.bmp) est un format inventé par Microsoft pour son propre système et qui est aujourd'hui largement répandu sur toutes les plates-formes. J'ai chosi d'utiliser ce format pour ne pas perdre de temps à programmer un décodeur pour un autre format (ou à comprendre comment JAVA pouvait me permettre de faire cela). En effet, le format bmp ne supporte que quelques formes de compressions simples. Il existe même une option non-compressée où les données sont stockées telles quelles, après un header et une table de couleurs. Ce choix m'a permis de disposer rapidement de matrices de test pour le reste du programme. J'ai donc choisi de ne supporter que le format non compressé en 8 bit par pixels (en niveaux de gris) : ce format m'offrait toute la simplicité souhaitée.

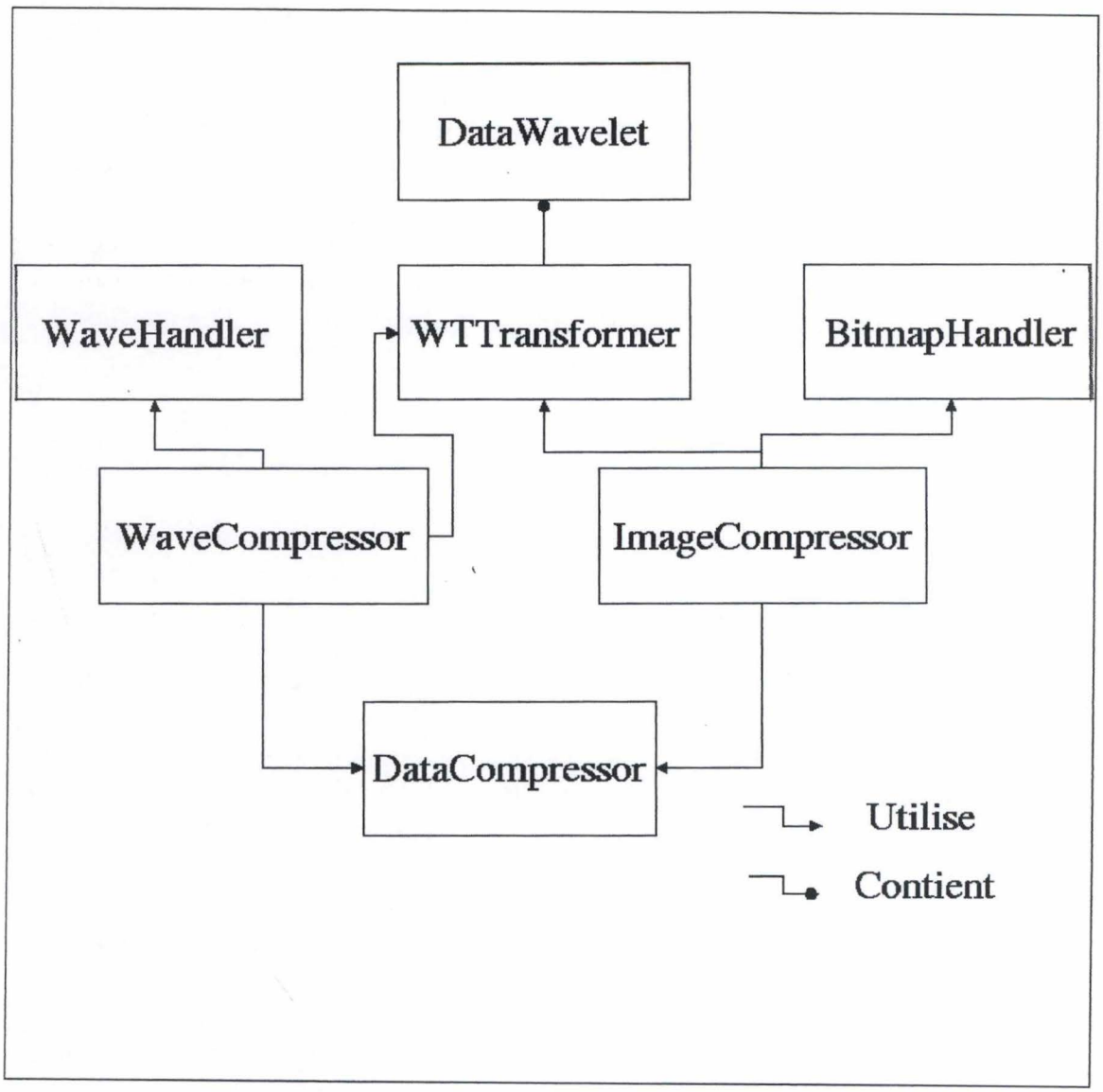


FIG. 3.3 – *Shéma de classes du package WaveletTools.*

Mon objectif en écrivant la classe `BitmapHandler` était de fournir au programmeur un ensemble de fonctions lui permettant de manipuler facilement les fichier .bmp *non-compressés en 8bpp*. Ainsi, la classe présente un certain nombre d'attributs qui sont prévus pour reçevoir chacun une partie de l'information du fichier. J'ai donc toute une série d'attributs correspondant aux différents bytes du header, à la table des couleurs et aux données brutes de l'image (`rasterData`). Comme je l'ai expliqué plus haut, ces attributs sont privés, la classe est autonome et contrôle fermement les accès aux données. En effet, pour pouvoir utiliser les fonctionnalités de `BitmapHandler`, il faudra passer par les fonctions dédiées aux accès aux données. Ceci répond à un principe sain de programmation OO.

Pour bien appréhender le fonctionnement interne de cette classe, nous allons examiner les principales fonctions qu'elle propose. Les autres classes fonctionnant sur le même modèle, je ne produirai cette approche détaillée que pour celle-ci.

- le constructeur `BitmapHandler()`.
- `setFileParameters(String file)` : cette fonction met les valeurs des attributs privés à celles des paramètres correspondants trouvées dans le fichier 'file' qui doit être un .bmp.
- `extractBMPData(String file)` : cette fonction extrait les données brutes d'un fichier .bmp (la matrice de pixels) et les stocke dans un tableau privé (rasterData).
- `getRasterData()` : cette fonction renvoie le tableau des données brutes.
- `setRasterDta(int[] data)` : cette fonction place les valeurs de 'data' dans le tableau rasterData.
- `composeBMP(String file)` : cette fonction compose un fichier 'file'.bmp avec les paramètres courants et les données de rasterData.
- Il reste quelques autres fonctions plus anecdotiques dont l'utilité est moins évidente.

L'utilisation de `BitmapHandler` se fait donc dans les deux sens : extraire des données d'un fichier .bmp pour les passer à la suite du programme et se servir de données pour composer un fichier .bmp (avec l'image reconstruite par exemple). Mon programme ne comprend donc pas d'afficheur d'image, je composais des fichier .bmp avec les données intéressantes et les affichait avec un viewer quelconque (dans ce cas précis, GQview).

2. `WTTransformer` : La classe effectuant la Transformée en Ondelettes

   Cette classe est exclusivement consacrée à la Transformée en Ondelettes. Elle fonctionne comme la précedente, i.e. elle comprend un ensemble d'attributs privés qui permettent d'effectuer la transformée et les fonctions nécessaires pour garnir ceux-ci et en récupérer les valeurs. L'utilisation de cette classe se fait donc avec la méthode suivante :

   (a) Garnir les attibuts avec les valeurs d'entrée via les fonctions dédiées

(b) Effectuer la transformée sur ces attributs

(c) Récupérer les valeur de sortie via d'autres fonctions dédiées

La manière la plus simple d'utiliser cette classe est d'utiliser le constructeur standard avec toute une série de paramètres qui serviront à initialiser les variables de travail, d'appeller la fonction `wnt()` et de récupérer les résultats avec la fonction `getOut()`. Toutefois, le progammeur averti dispose de toutes les fonctions nécessaires pour utilliser cette classe comme il l'entend (avec toutefois le risque de ne pas avoir les résultats escomptés s'il oublie une étape ou l'autre).

La fonction principale est évidement celle qui effectue la Transformée en Ondelettes : `wtn`. L'algorithme que j'ai utillisé ici est celui qui est employé dans les routines du livre Numerical Recipies in Fortran 77 ([33]) et qui est disponible sur le web. Cet algorithme diffère légèrement de celui décrit par Mallat parce qu'il reprend les optimisations qui ont été proposées à la suite de son article. Néanmoins, le lecteur pourra constater que les calculs ne diffèrent guère. La différence principale étant que cette approche se fait d'un point de vue matriciel.

La fonction `wtn()` effectue l'algorithme pyramidal alors que la fonction `pwt()` en effectue une étape. La fonction `pwtset` actualise les paramètres de transformation. Ces paramètres sont essentiellement constitués des coefficients des filtres utilisés pour effectuer la transformée. L'algorithme ne supporte que les filtres symétriques et, pour l'instant, uniquement ceux de Daubechies (4, 12 et 20 coefficients) et Haar.

L'utilisateur peut évidemment complètement configurer la classe pour effectuer la tranformée comme il l'entend.

Toutes les données de filtres utilisés pour effectuer les calculs sont regroupées dans la classe `DataWavelet`.

3. `DataWavelet` : regroupement des données des filtres

   Comme je l'ai indiqué plus haut, cette classe regroupe toutes les données des filtres que l'on peut employer pour effectuer la Transformée en Ondelettes. Elle fournit également une fonction `getData` qui permet de récupérer les coefficients que l'on désire. Cette classe contient également les coefficients d'autres filtres qui sont prévus pour servir lorsque j'aurai changé l'algorithme de transformation pour qu'il supporte les filtres non-symétriques.

4. `DataCompressor` : compression et filtrage

   Cette classe est consacrée au filtrage des données en fonction d'un seuil donné et à la compression effective par codage EZW et arithmétique (ces deux fonctionnalités n'étant malheureusement pas encore disponibles).

   Plus que toute autre, cette classe est complètement configurable. Sa fonctionnalité principale est de filtrer les coefficients, c'est-à-dire de les mettre à zéro

en fonction d'un certain seuil. L'utilisateur peut entièrement paramètrer la manière de procéder. En effet, il peut configurer

(a) Le mode de comparaison des coefficients par rapport au seuil : en valeur absolue ou relative.

(b) Le type de comparaison par rapport à un nombre réel qu'il fourni x
- Comparaison simple : $c_{i,j} < x \Rightarrow c'_{i,j} = 0$
- Comparaison par rapport à la moyenne des coefficients $mean$: $c_{i,j} < mean * x \Rightarrow c'_{i,j} = 0$
- Comparaison par rapport au maximum des coefficients Max : $c_{i,j} < Max * x \Rightarrow c'_{i,j} = 0$
- Comparaison par rapport à l'écart maximal des coefficients $M = Max - min$: $c_{i,j} < M * x \Rightarrow c'_{i,j} = 0$
- Comparaison par rapport à l'écart entre la moyenne et le minimum des coefficients $m = mean - min$: $c_{i,j} < m * x \Rightarrow c'_{i,j} = 0$
- Suppression approximative de $x\%$ des coefficients. Pour cette option, j'ai utilisé une approche rapide mais approximative. En effet, la méthode consiste à trier les coefficients par ordre croissant (via une routine de QuickSort disponible gratuitement dans la documentation de JAVA) et à prendre comme valeur de seuil le coefficient qui se trouve à $x\%$ de la longueur totale en partant de la fin du tableau. J'utilise ensuite cette valeur pour une comparaison simple. J'ai utilisé cette méthode avec beaucoup de suspiscion pour aller vite au début de mes essais et, comme les résultats effectifs se sont révélés très positifs (le poucentage d'éléments mis à zéro est toujours très proche de celui désiré), j'ai décidé de la conserver dans le programme.

(c) Le mode d'information de l'utilisateur :
- Aucun feed-back à l'utilisateur
- Les informations de filtrage seront imprimées à l'écran
- Les informations de filtrage seront imprimées dans un fichier dont le nom est `FilteringInfo.dat`

Les informations de filtrage sont :
- la valeur x fournie par l'utilisateur
- La moyenne des coefficients
- Le minimum des coefficients
- Le maximum des coefficients
- L'écart min-max
- L'écart min-mean
- L'écart quadratique moyen des coefficients par rapport à leur moyenne
- Le pourcentage de coefficients mis à zéro
- Le nombre de coefficients mis à zéro
- Le mode de comparaison

5. `ImageCompressor` : classe globale. Cette classe pourrait être vue comme un

exemple d'utilisation des classes précedentes. Elle propose une fonction `CompressImage` qui prend comme argument le nom d'un fichier de paramètres et qui utilise toutes les fonctionnalités des classes du package pour appliquer la Transformée en Ondelettes à une image, filtrer les coefficients et recomposer une image à partir de la matrice quantifiée. En outre, cette méthode permet de pratiquer la transformée par blocs dont on peut spécifier les dimensions. Cette option ne m'a servi qu'à quelques tests.

6. Encore quelques classes dédiées à la compression de sons mais dont nous parlerons plus loin.

3.2.2 Le programme FORTRAN90

Pour faire ce programme, j'ai utilisé une nouvelle fonctionnalité du FORTRAN90 : les modules. Ces structures permettent de regrouper des variables ou des fonctions un peu à la manière des Classes. Ces modules font d'ailleurs le pendant aux classes en JAVA.

La figure 3.4 montre le schéma du programme.

3.3 Résultats

Je vais présenter, dans cette section, les résultats obtenus par les programmes présentés plus haut. Il s'agit ici des résultats du programme FORTRAN mais ceux-ci sont identiques à ceux du programme JAVA.

On ne semble pas pouvoir y échapper lorsqu'on s'intéresse au traitement d'images, Lena est une image test de référence. Cette photo, qui est celle de Miss Mars de Playboy en 1972, s'est durablement installée dans la littérature. Aujourd'hui, plus aucun article ne présente ses résultats sans Lena, elle sert même de référence pour procéder à des comparaisons entre schémas de codage. Je sacrifie donc à la tradition et présente les résultats que j'ai obtenu avec cette image.

La figure 3.5 montre la photo originale, sa Transformée en Ondelettes (couleurs inversées) et la reconstruction de l'image en conservant 100% des coefficients. Nous constaterons tout d'abord la qualité de cette reconstruction : il est parfaitement impossible de distinguer l'original de celle-ci.

Intéressons-nous à la Transformée. Comme je l'avais dit tout à l'heure, l'algorithme est légèrement différent de celui utilisé par Mallat. Cependant, on distingue très bien les différentes zones mentionnés par celui-ci lors de sa description de la Résolution en Ondelettes. Ainsi, l'image est tout d'abord divisée en quatre zones. On aperçoit dans la zone en bas à droite les hautes fréquences verticales et horizontales. Dans les zones bas-gauche et haut-droite, les hautes fréquences horizontales et verticales. Enfin, dans la partie haut-gauche, on constate que ce schéma se reproduit mais dans une autre gamme de fréquence. La décomposition pyramidale apparaît ainsi clairement.

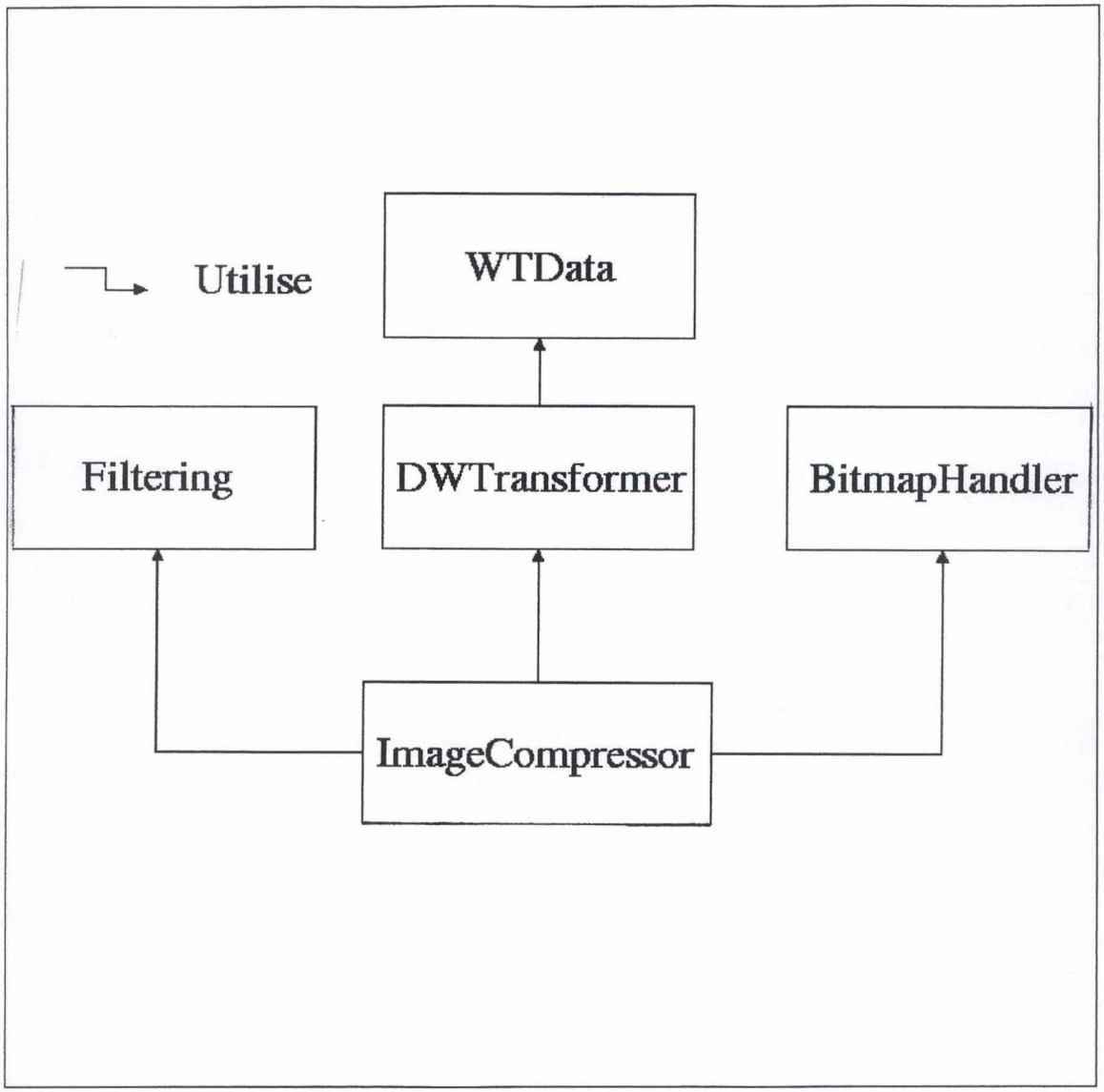


FIG. 3.4 – *Shéma du programme en FORTRAN90.*

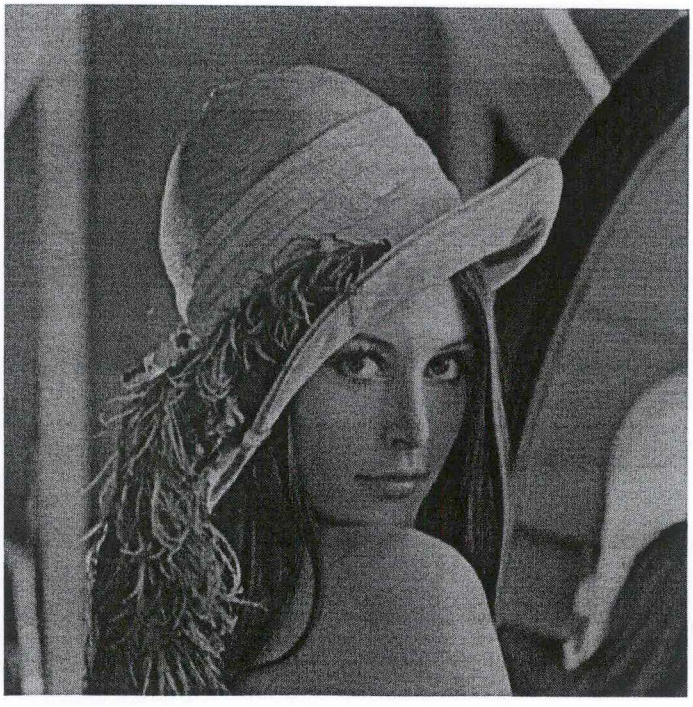


FIG. 3.5 – *Photo de Lena originale, sa Transformée en Ondelettes Discrète (fortement contrastée) et l'image reconstruite*

FIG. 3.6 – *Lena reconstruite à partir de la Transformée en Ondelettes Discrète (TOD) filtrée à 50%.*

Examinons à présent l'effet de la mise à zéro des coefficients. Les figures 3.6 ,3.7 et 3.8 donnent respectivement les images reconstruites à partir des Transformées en Ondelettes Discrètes filtrées à 50%, 80% et 90%. On constatera que la perte de qualité à 90% est encore largement négligeable. Sur chaque Transformée en Ondelettes Discrète, constatons qu'il s'agit bien des hautes fréquences que nous supprimons majoritairement.

Pour terminer, les figures 3.9 , 3.10 et 3.11 montrent les reconstructions à partir de Transformées en Ondelettes Discrètes filtrées à 95%, 99% et 99,9%. Il est remarquable de constater que les lignes directrices de l'image restent visibles, même à ces très forts taux de compression. Notons également que l'on aperçoit que c'est bien autour des bords que ce concentrent les défauts et qu'ils s'adaptent à la couleur de l'image.

J'ai également effectué d'autres tests. En guise d'aperçu, je donne ici une des images que j'ai utilisées; figures 3.12 3.13 ,3.14 et 3.15 , 3.16 , 3.17 et 3.18 . On les voit ici aux mêmes taux de mise à zéro que l'image de Lena. Tout comme pour Lena, on notera la qualité des images reconstruites, même aux grands taux de compression.

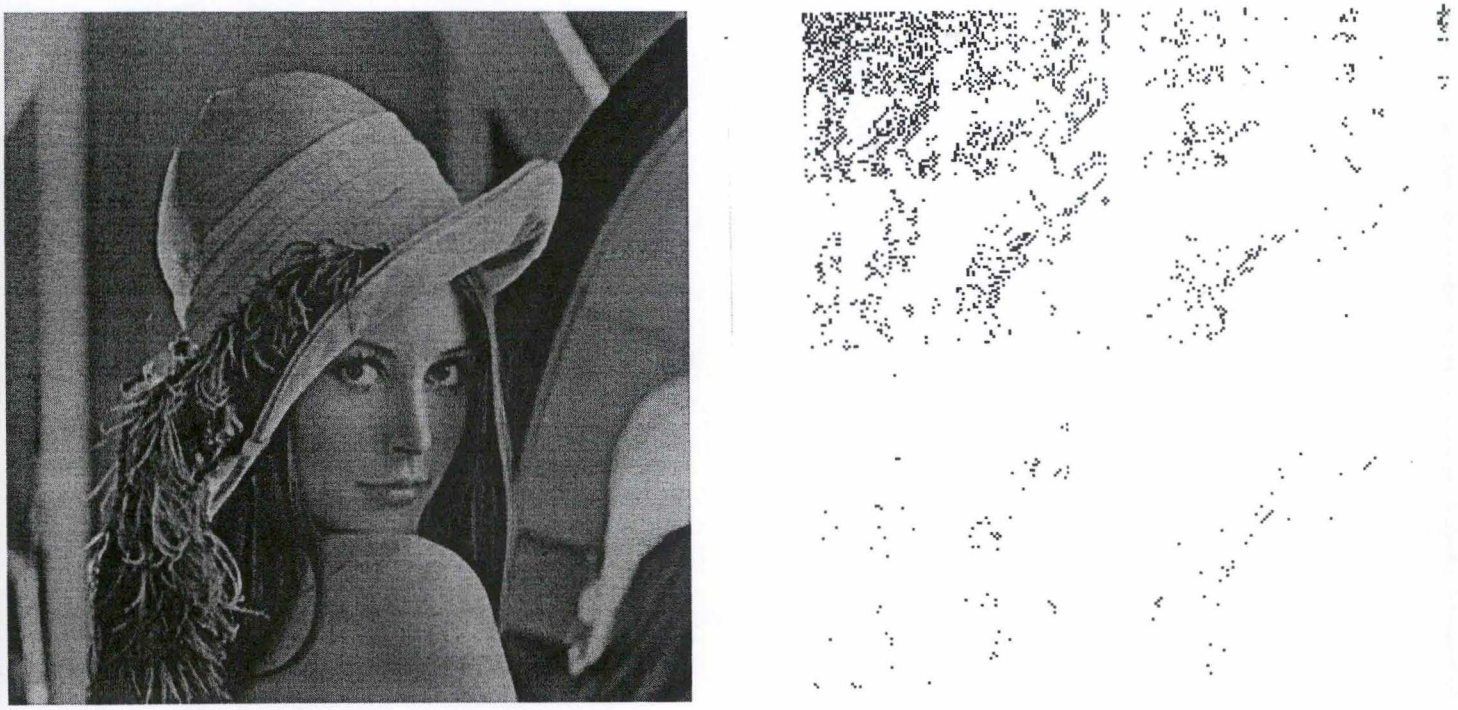


FIG. 3.7 – *Lena reconstruite à partir de la TOD filtrée à 80%*

FIG. 3.8 – *Lena reconstruite à partir de la TOD filtrée à 90%*

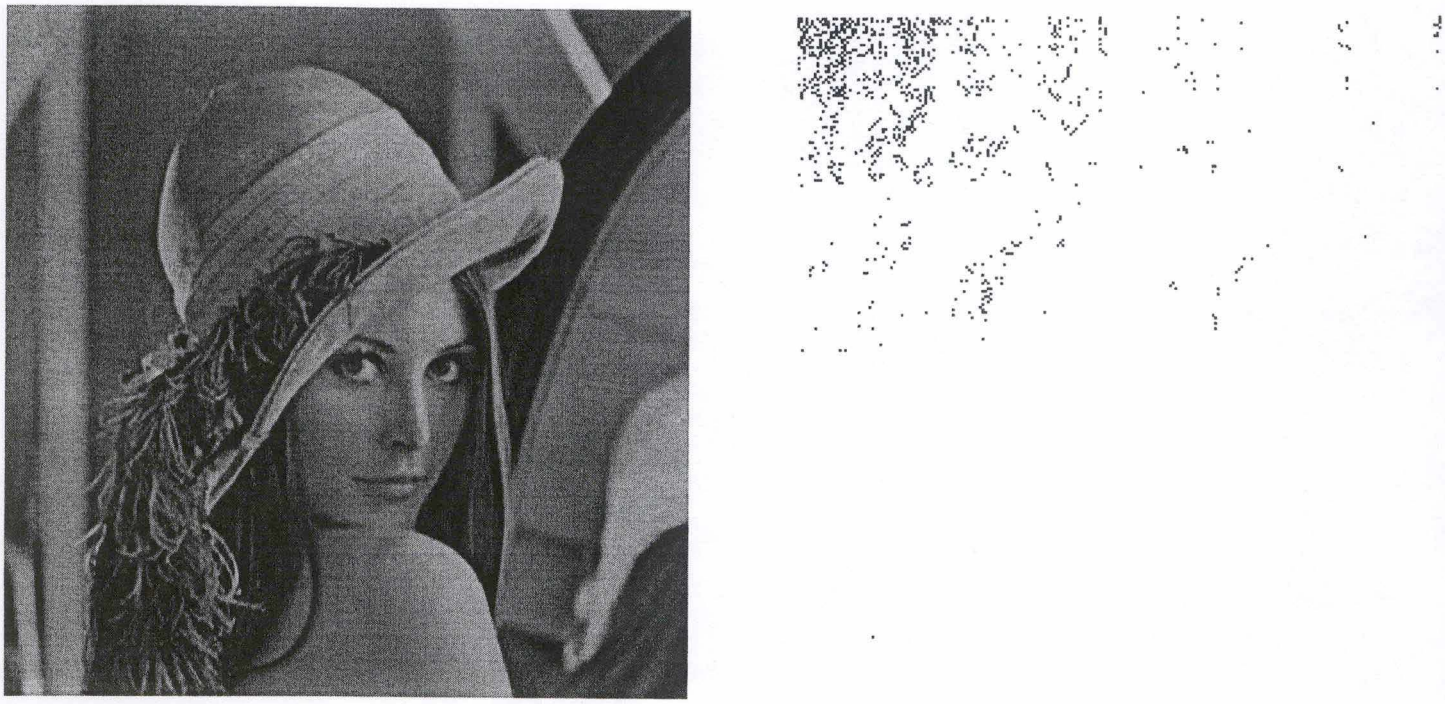


FIG. 3.9 – *Lena reconstruite à partir de la TOD filtrée à 95%.*

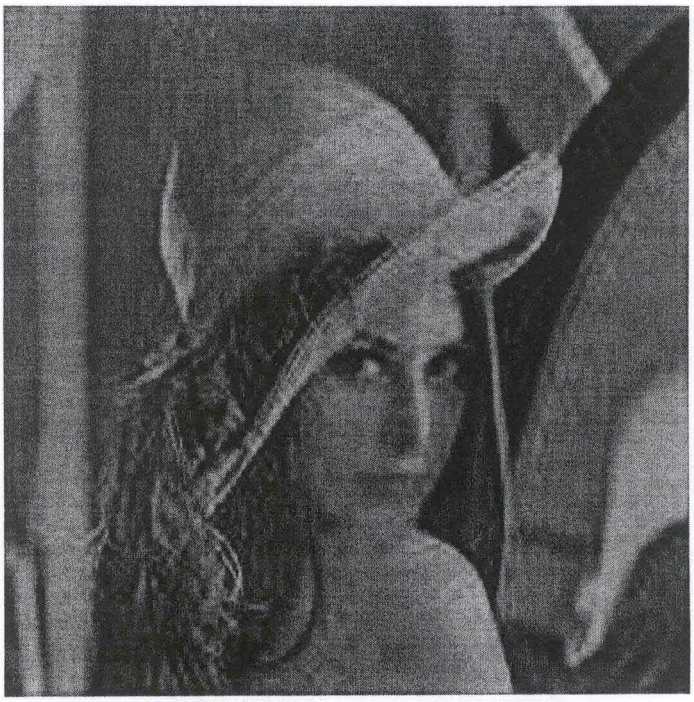


FIG. 3.10 – *Lena reconstruite à partir de la TOD filtrée à 99%.*

FIG. 3.11 – *Lena reconstruite à partir de la TOD filtrée à 99,9%.*

3.4 Algorithmes de codage de la représentation en Ondelettes

Dans cette section, nous nous intéresserons aux algorithmes qui permettent de coder la Représentation en Ondelettes. Il s'agit de l'étape précédant le codage entropique, comme nous l'avons mentionné au point 3.1.

Seul l'algorithme fondateur, proposé par Jérôme Shapiro en 1993 et baptisé EZW sera présenté dans ses principes fondamentaux [41]. Ce choix est fait en raison du caractère fondateur de cet algorithme. En effet, les autres algorithmes de codage développés à ce jour sont toujours plus ou moins des raffinements des idées proposées par Shapiro. On dénombre aujourd'hui plus de 10 algorithmes différents, [2, 41, 40, 34, 39, 37, 50, 42, 38, 36], dont les performances sont axées sur l'un ou l'autre aspect du traitement d'image. Cette diversité montre le foisonnement intense qui se fait autour de ce domaine mais aussi l'absence de standard, ce qui explique sans doute pourquoi la compression en Ondelettes n'est pas encore répandue auprès du grand public malgré ses indéniables qualités. La standardisation des méthodes est d'ailleurs à l'ordre du jour et a fait récemment l'objet de publications([46, 45]).

La majorité du texte de cette section est principalement tiré de l'article de Shapiro [41].

FIG. 3.12 – *Photo originale de tigre, sa Transformée en Ondelettes Discrète (fortement contrastée) et l'image reconstruite*

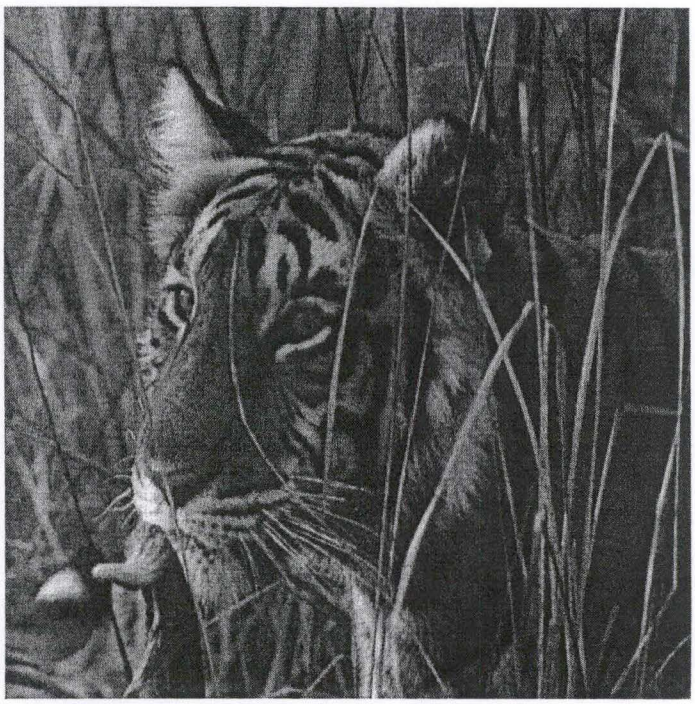


FIG. 3.13 – *tigre reconstruit à partir de la Transformée en Ondelettes Discrète (TOD) filtrée à 50%.*

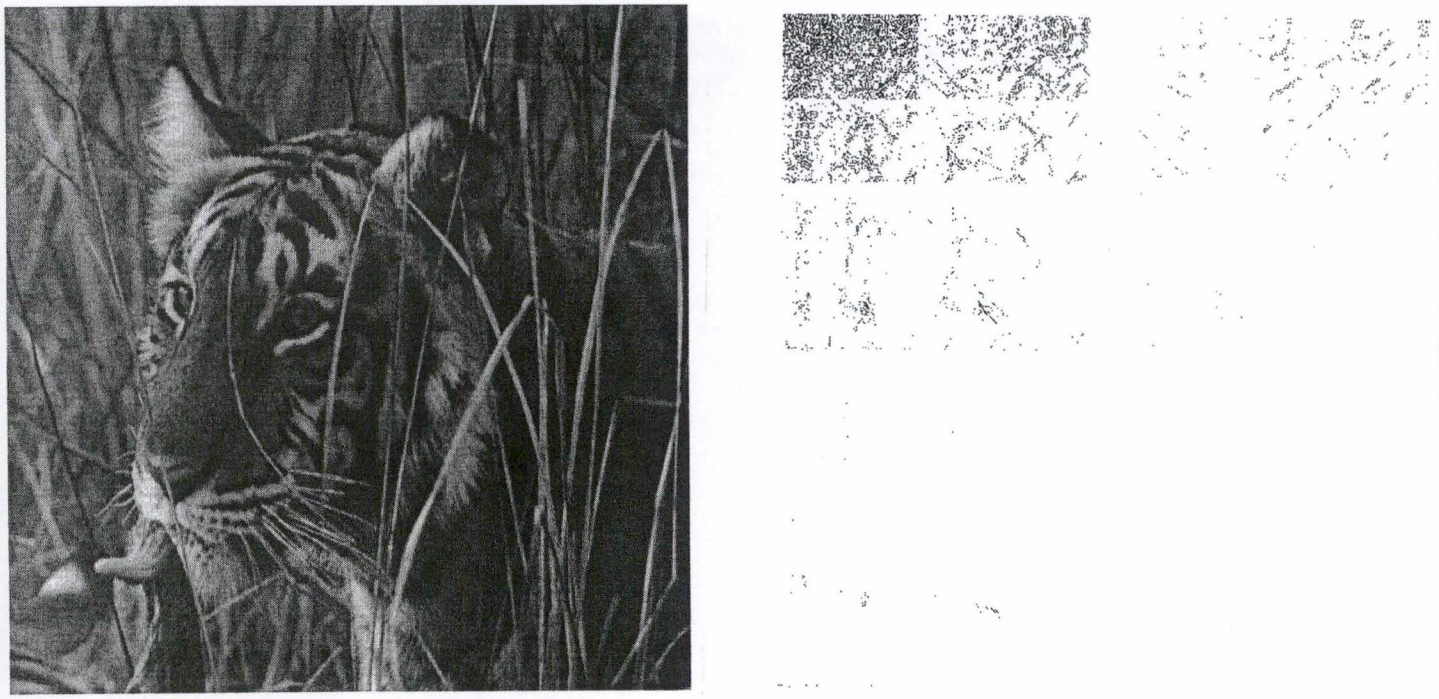


FIG. 3.14 – *tigre reconstruit à partir de la TOD filtrée à 80%.*

FIG. 3.15 – *tigre reconstruit à partir de la TOD filtrée à 90%.*

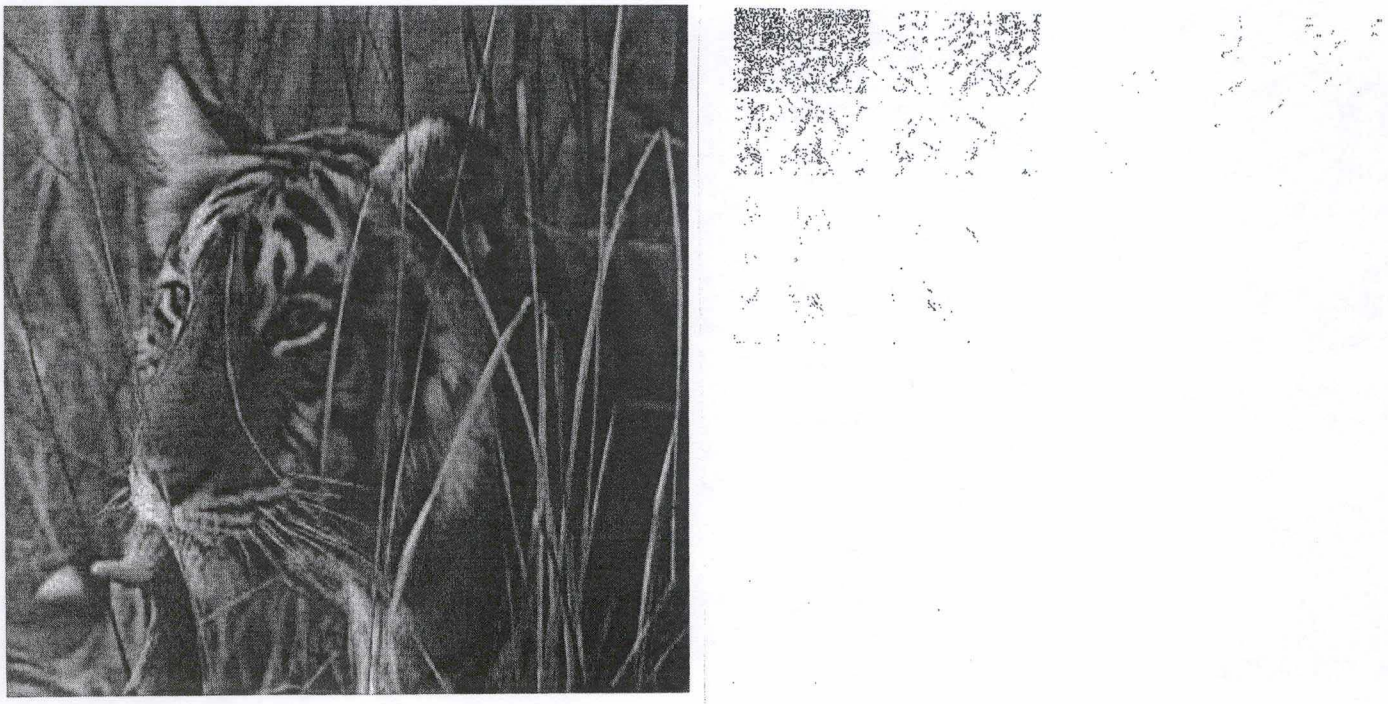


FIG. 3.16 – *tigre reconstruit à partir de la TOD filtrée à 95%*

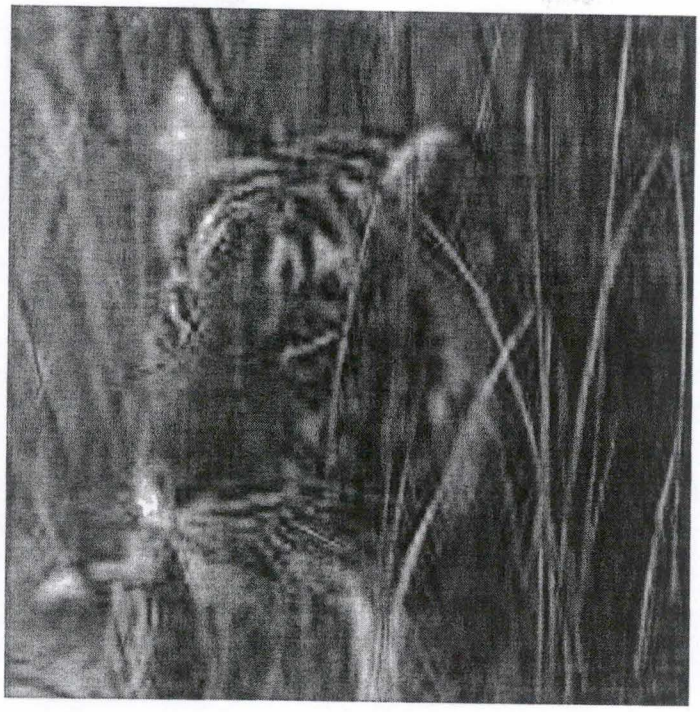


FIG. 3.17 – *tigre reconstruit à partir de la TOD filtrée à 99%.*

FIG. 3.18 – *tigre reconstruit à partir de la TOD filtrée à 99,9%.*

3.4.1 Algorithme EZW

Comme on peut le voir dans l'annexe C, la Transformée en Ondelettes Discrète se prête bien à la transmission progressive d'images. Les objectifs de Shapiro lorsqu'il écrivit son article étaient, non seulement de proposer un shéma de codage efficace, mais aussi de produire un code adapté à ce type de transmission (Embedded code). Ainsi, Shapiro définissait son code comme une suite de bits ordonnés par ordre d'importance, permettant à l'encodeur de s'arrêter à n'importe quel moment du processus en fonction par exemple d'un objectif de compression ou de taux de distorsion (on peut entre autres diriger l'encodeur en fonction du nombre de bits par pixel). De la même façon, le décodeur peut arrêter son travail dès que ses objectifs sont atteints en terme de qualité(s) de reconstruction. Shapiro compare cela à la représentation binaire des nombres réels. Ces nombres peuvent être représentés par une suite de bits. Chaque bit ajouté à droite augmente la précision sur le nombre représenté. Lorsqu'une précision suffisante a été atteinte, on peut arrêter le codage ou le décodage.

Pour répondre à ces objectifs, l'algorithme EZW se compose de plusieurs parties distinctes :

- Une Transformée en Ondelettes donnant une représentation compacte de l'image.
- Un codage utilisant la notion de Zerotree donnant une représentation compacte de la carte des coefficients significatifs (signifiance map).
- Les approximations sur la qualité de l'image pour atteindre différents taux de compressions afin de produire le code présentant les qualités décrites ci-dessus sont alors faites en deux étapes :
  - approximations successives
  - prioritisation pour l'ordonnancement des bits
- Le travail se termine par un codage arithmétique.

Nous ne nous intéresserons ici qu'au codage Zerotree qui constitue l'élément original repris par les codages ultérieurs.

Les images naturelles sont principalement composées de deux éléments : des surfaces de pixels fortement corellés qui sont les "tendances" de l'image, les basses fréquences spatiales et des "détails" dont l'importance est très grande au niveau perceptuel mais dont la contribution à "l'énergie" de l'image est faible. Les codeurs du type JPEG fonctionnent avec la Transformée en Cosinus Discrète, décomposent l'image dans une représentation où chaque coefficient correspond à une surface fixe et à une bande de fréquence fixe qui sont les mêmes pour tous les coefficients (cfr chapitre 1). Le problème est alors que les bords nécessitent énormément de coefficients non-nuls en dépit de leur "faible importance" dans l'image. Aux taux de compression élevés, le nombre de bits associés aux tendances devient trop grand par rapport à ceux associés aux détails et il en résulte des distorsions du type blocking artifac.

Nous avons vu comment la représentation en Ondelettes stocke les images : à partir d'une approximation à faible résolution en y ajoutant les niveaux de détails des

FIG. 3.19 – *Représentation en Odelettes orthogonales de Lena.*

résolutions supérieures (figure 3.19). La figure 3.20 nous donne une représentation schématique à deux niveaux. Comme nous le savons déjà, la zone HH_1 correspond aux hautes fréquences verticales et horizontales, les zones LH_1 et HL_1 correspondent aux hautes fréquences horizontales et verticales tandis que le quatrième quart correspond à une approximation de résolution inférieure, elle aussi divisée en bandes de fréquences orientées. Comme nous l'avons déjà vu, le problème de la quantification de cette représentation se situe dans la nécessité de coder la position des coefficients significatifs. Ces coefficients significatifs sont déterminés par comparaison vis-à-vis d'un seuil fixé, à la manière habituelle : $c_{ij\in}$ coefficients significatifs $\Leftrightarrow |c_{ij}| \geqslant T$ où T représente le seuil (Threshold).

Le concept d'arbre Zerotree est basé sur l'hypothèse que si un coefficient à une résolution faible est non-significatif, alors il existe une forte probabilité que tous les coefficients de la même orientation mais aux résolutions plus grandes soient non-significatifs également. Cette hypothèse est largement justifiée par l'étude empirique des Représentation en Ondelettes d'images naturelles.

Voyons cela : dans la Représentation en Ondelettes telle que l'a définie Mallat, on peut faire correspondre chaque coefficient à une résolution donnée à un ensemble de coefficients aux résolutions plus hautes (à l'exception, bien entendu, des coefficients des bandes de premier niveau : HH_1, LH_1, HL_1). On peut voir à la figure 3.21 la manière dont cette correspondance s'organise. De façon assez naturelle, on imagine une structure d'arbre. Les coefficients de résolution basse sont alors les noeuds parents

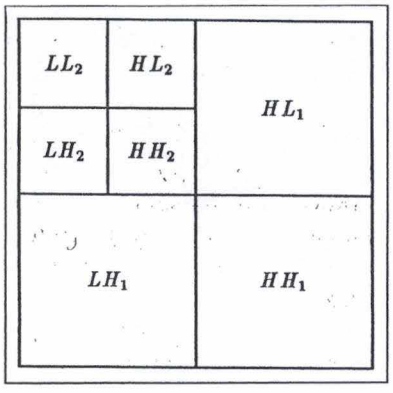


FIG. 3.20 – *Représentation schématique à deux niveaux de la Représentation en Ondelettes d'une image.*

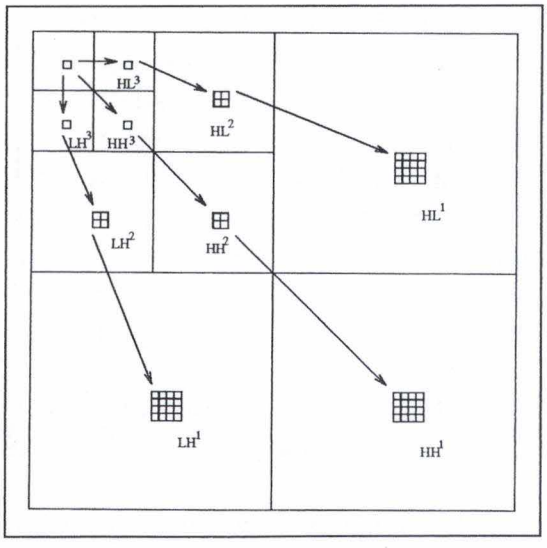


FIG. 3.21 – *Structures d'arbres dans la Représentation en Ondelettes.*

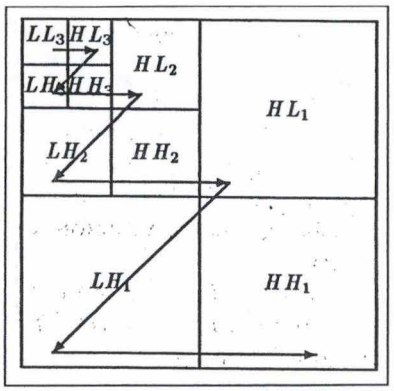


FIG. 3.22 – *Méthode d'exploration des bandes de fréquences. Les coefficients parents doivent être examinés avant leurs enfants. Tous les coefficients d'une bande sont examinés avant de passer à la suivante.*

et les coefficients de même orientation aux résolutions plus élevées sont les noeuds enfants et les feuilles. Sur cette figure, on peut voir un arbre issu d'un coefficient situé en HH_3 dans lequel chaque parent donna quatre enfants au niveau supérieur. Les racines des arbres, situées dans l'approximation à plus faible résolution, donnent chacune trois enfants dans les bandes de fréquence adjacentes (en fait, le départ de trois arbres "normaux").

Le codage se base sur la structure de ces arbres présents naturellement dans une Représentation en Ondelettes. L'algorithme va procéder au passage en revue des coefficients dans un ordre tel qu'un enfant ne puisse être examiné avant ses parents. Cela signifie donc qu'il faut explorer les bandes de basses fréquences avant celles de hautes, comme indiqué à la figure 3.22. Sur ce shéma, le passage d'une flèche par une bande de fréquence indique l'examen de tous les coefficients de cette bande.

En fonction d'un seuil T, on dira que c_{ij} appartient à un arbre Zerotree si tous ses descendants sont non-significatifs vis-à-vis de T. Le coefficient $c =_{ij}$ sera racine d'un tel arbre si c_{ij} n'est pas descendant d'une racine d'arbre Zerotree précédemment trouvée. Donc, même si notre hypothèse est justifiée, dans beaucoup de cas, il est possible qu'un coefficient non-signficatif possède l'un ou l'autre descendant significatif.

En conséquence, nous pouvons encoder la structure de la Représentation en Ondelettes (significance map) avec trois symboles seulement ! Ceux-ci sont :

1. racine d'arbre Zerotree : dont tous les descendants seront non-significatifs.
2. zéro isolé : coefficient non-significatif dont certains descendants sont significatifs
3. coefficient significatif

De plus, la structure des bandes de premier niveau sera encodée avec deux symboles puisque le symbole racine d'arbre Zerotree ne sera pas utilisé.

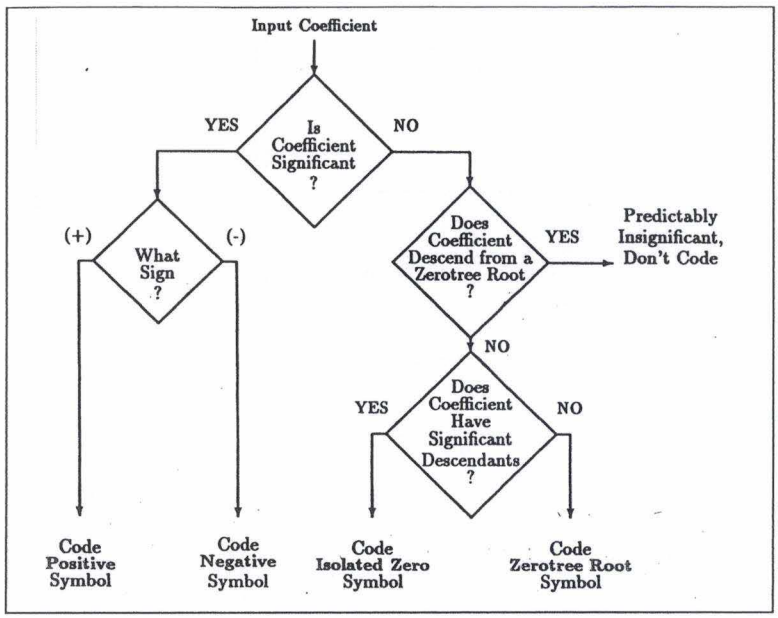


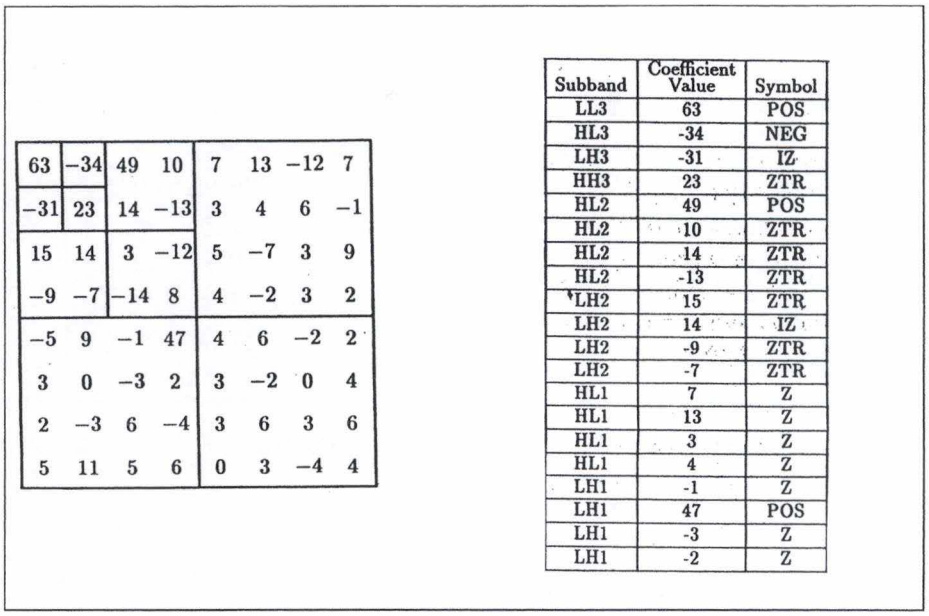
FIG. 3.23 – *Méthode de décision pour la construction du code.*

Dans son algorithme, Shapiro code également le signe des coefficients significatifs parce que cette information lui sert dans les quatre étapes ultérieures de codage. En pratique, nous avons donc quatre symboles :

1. racine de Zerotree
2. zéro isolé
3. coefficient significatif positif
4. coefficient significatif négatif

On peut voir à la figure 3.23 la méthode de décision pour construire le code à partir de l'examen des coefficients. Une fois celui-ci terminé, le code peut être ensuite compressé via un algorithme entropique. La figure 3.24 montre le codage d'une petite matrice de coefficients.

Il existe un parallèlisme entre la structure d'arbre de Zerotree et le symbole de fin de bloc (EOB) utilisé en condage JPEG. Ce symbole indique la fin de codage d'un bloc : tous les autres éléments de la matrice 16x16 (ou 8x8) considérée sont nuls. On peut immédiatement voir l'avantage énorme que les arbres Zerotree possèdent sur le symbole EOB. En effet, ce symbole représente la "non-significance" d"un ensemble de coefficients dont le nombre vaut au plus 256 (ou 64). Au contraire, la Transformée en Ondelettes Discrète est une représentation hiérarchique de l'image. Pour une taille de bloc 16x16, il existe exactement une résolution dans la Représentation en Ondelettes à cette échelle. Si on trouve une racine d'arbre Zerotree dans cette bande, celui-ci correspondra à la même surface que le symbole EOB. Par contre, si on trouve une racine d'arbre Zerotree à résolution plus faible, l'ensemble de coefficients



63	−34	49	10	7	13	−12	7
−31	23	14	−13	3	4	6	−1
15	14	3	−12	5	−7	3	9
−9	−7	−14	8	4	−2	3	2
−5	9	−1	47	4	6	−2	2
3	0	−3	2	3	−2	0	4
2	−3	6	−4	3	6	3	6
5	11	5	6	0	3	−4	4

Subband	Coefficient Value	Symbol
LL3	63	POS
HL3	-34	NEG
LH3	-31	IZ
HH3	23	ZTR
HL2	49	POS
HL2	10	ZTR
HL2	14	ZTR
HL2	-13	ZTR
LH2	15	ZTR
LH2	14	IZ
LH2	-9	ZTR
LH2	-7	ZTR
HL1	7	Z
HL1	13	Z
HL1	3	Z
HL1	4	Z
LH1	-1	Z
LH1	47	POS
LH1	-3	Z
LH1	-2	Z

FIG. 3.24 – *Codage d'une petite matrice de coefficients.*

non-significatifs représenté est alors beaucoup plus grand et la surface représentée beaucoup plus importante. De plus, si on ne trouve pas de racine de Zerotree à cette résolution, la recherche à des résolutions plus fines représente une manière plus intelligente et hiérarchique de rechercher des zones non-significatives à l'intérieur de la surface représentée par le symbole EOB. Ainsi, l'approche Zerotree est très efficace pour isoler des détails significatifs intéressant ou éliminant immédiatement de larges régions non-significatives.

Dans la suite de l'algorithme EZW, le codage de la structure de la Représentation en Ondelettes (signifiance map) est exploité pour atteindre les objectifs que j'ai mentionné tout à l'heure. Nous nous en tiendrons à cette notion d'arbre Zerotree qui était l'idée essentielle de la méthode et qui fut largement réexploitée.

3.4.2 Autres algorithmes de codage basés sur la Transformée en Ondelettes Discrète

Comme je l'ai déjà mentionné, les algorithmes de codage se multiplient comme des petits pains un jour de sermon sur la montagne. Comme ceux-ci ne sont pas directement liés à mon travail, je ne vais pas les présenter ici en détails (le lecteur intéressé pourra toujours se référer à la bibliographie).

Que l'on sache simplement que certains sont des évolutions du travail de Shapiro avec un raffinement du modèle des images naturelles et de la technique de structuration des arbres : MRWD [40] et EPWIC [34].

D'autres ont portés leurs efforts sur la technique de codage : Xiong [50].

Codeur	Bits par Pixel								
	0.008	0.016	0.0031	0.063	0.125	0.25	0.50	1.0	2.0
JPEG	-	-	17.95	21.92	28.24	31.42	34.84	37.95	41.62
EZW	21.03	22.99	25.01	27.46	30.26	33.32	36.46	39.79	44.64
EPWIC-1	21.26	23.35	25.41	27.61	30.18	33.04	35.99	39.27	44.07
EPWIC-2	21.68	23.54	25.70	28.03	30.85	33.78	37.15	40.34	45.06
MRWD-1	-	-	-	-	-	34.12	37.18	40.33	-
MRWD-2	-	-	-	-	-	33.90	37.01	40.20	-
Xiong	-	-	-	-	-	34.33	37.36	40.52	-
SPIHT	22.07	23.95	25.95	28.36	31.10	34.12	37.23	40.43	45.11

TAB. 3.1 – Tableau des performances comparées des algorithmes de compression en terme de rapport signal sur bruit (db)

D'autres ont élaboré de nouveaux concepts inspirés des travaux précédents : SPIHT [39, 37, 38].

Pour finir, certains utilisent une évolution de la Transformée en Ondelettes qui est appellée "Wavelet packet" : Wpkt [42].

Il est difficile de comparer les performances de ces algorithmes en raison de leurs diversités et de la présentation des résultats obtenus. Le tableau 3.1 donne une comparaison en terme de rapport signal sur bruit (db) pour différents systèmes avec l'image Lena, sur base des informations que j'ai pu rassembler.

En guise de conclusion de cette partie, nous dirons que, jusqu'à présent, ce sont les algorithmes de Xiong et SPIHT qui sont les meilleurs pour la compression d'image (en tout cas à ma connaissance). On constatera également que tous ces algorithmes présentent des performances supérieures à la norme JPEG (ces performances sont réellement très supérieures puisque le rapport signal sur bruit est donné en db !)

Conclusion

Nous voilà au terme de notre voyage au pays des Ondelettes, revoyons ensemble quelles en ont été les étapes.

Nous avons tout d'abord rencontré la Transformée en Ondelettes Continue. Nous avons vu quels étaient ses avantages sur les outils usuels de traitement du signal. Nous avons montrés les principaux résultats liés à la décomposition sur une base d'ondelettes et ainsi pu apprécier les potentialités énormes de cette méthode.

L'étape suivante, la plus longue, a été la découverte de la Transformée en Ondelettes Discrète, ou de la Représentation en Ondelettes Orthogonales. Nous avons montré comment on peut définir un modèle très élégant permettant de représenter une image par une approximation à faible résolution et un ensemble des différences de détails aux résolutions plus élevées. Nous avons construit pas à pas l'algorithme pyramidal qui permet d'obtenir cette représentation. Nous avons interprété celle-ci comme une décomposition de l'image initiale en bandes de fréquene indépendantes et spacialement orientées. Nous avons vu que la matrice de pixels originale pouvait être retrouvée via un algorithme inverse de même type que le premier. Nous avons enfin vu une modélisation simple mais efficace de l'histogramme de répartition des coefficients de la Représentation en Ondelettes Orthogonales.

Nous sommes ensuite passé voir comment fonctionnaient les programmes que j'ai écrit. Nous avons vu en détails la toolkit JAVA avec ses nombreuses potentialités pour manipuler des images au format bitmap Windows, effectuer la Transformée en Ondelettes Discrète grâce à un algorithme très efficace et filtrer les coefficients. Les résultats obtenus ont montré de façon éclatante que la Représentation en Ondelettes est particulièrement adaptée à la compression, même aux très hauts taux. Ces résultats ont été montré sur deux types d'images. Nous avonc également vu la toolkit FORTRAN90 dont les performances en terme de temps d'exécution sont très supérieures au JAVA.

Enfin, nous avons découvert les algorithmes de codage d'images compressées qui tirent parti de la structure très particulière des coefficients de la Représentation en Ondelettes, tant au niveau de leur position que de leurs propriétés statistiques. Ces algorithmes permettent d'obtenir des taux de compression inégalés à l'heure actuelle pour une distortion minimale. Nous avons particulièrement insisté sur le premier d'entre eux et dont ils descendent tous plus ou moins, l'algorithme EZW avec le concept d'arbre Zerotree.

Pour finir, j'ai mentionné un essai sur la compression sonore dont la littérature paraît fournie mais que je n'ai pas approfondi, ce sujet sortant du cadre de ce travail.

Je pense d'ailleurs que mon travail pourra servir à d'autres. La Transformée en Ondelettes est un outil particulièrement prometteur et ce dans de multiples domaines. Mon mémoire ouvre des pistes sur le traitement d'images et de sons qui peuvent être poursuivies. Le champ des applications de la Représentation en Ondelettes Orthogonales est encore largement en friche et toute une communauté s'active déjà à débrousailler de large pans. Je pense que l'Institut pourra désormais y prendre une part.

En guise de conclusion, je voudrais souligner que, d'un point de vue personnel, j'ai particulièrement apprécié de travailler sur une matière comme les Ondelettes. Les difficultés rencontrées au début de la réalisation de ce mémoire m'ont obligé à plonger dans une théorie ardue et dense mais tellement élégante (je pense particulièrement à la Représentation en Ondelettes de Stéphane Mallat). J'ai pu assister, et j'espère l'avoir fait faire aussi au lecteur, à la construction d'un algorithme à partir de concepts d'un abord très théorique. J'ai également pu appliquer cet algorithme pour obtenir des résultats concrets et impressionnants. Je pense que la Transformée en Ondelettes est promise à un bel avenir, le nombre de publications la concernant devient tout à fait énorme (comme j'ai d'ailleurs pu le constater au cours de cette année) et avoir participé, ne fut-ce qu'un peu, à cette vaste et foisonnante recherche a été une expérience dont je garderais un excellent souvenir.

Bibliographie

[1] Manduca A. and Said A. Wavelet compression of medical images with set partitionning in hierarchical trees.

[2] Mathieu P. Antonini M., Barlaud M. and Daubechies I. Image coding using wavelet transforms. *IEEE Transactions on Image Processing*, I(2) :205–220, April 1992.

[3] Corey Cheng. *Wavelet signal processing of digital audio with applications in electro-acoustic music.* PhD thesis, Dartmouth college, Hanover, New Hampshire, May 1996.

[4] Collectif. *Mathémathiques.* Encyclopédie des Sciences. Editions Alpha, 1976.

[5] J. Crowley. A representation for visual information. 1987.

[6] Ingrid Daubechies. *Ten Lectures on Wavelets.* S.I.A.M Press Philadelphia, sixth edition, 1992.

[7] Marshak A. Davis A. and Wiscombe W. Wavelet-based multifractal analysis of non-stationary and/or intermittent geophysical signals. August 1994.

[8] Ronald DeVore and Lucier Bradley. Wavelets. *Acta numerica.*

[9] Tim Edwards. Discrete wavelet transforms : theory and implementation. Stanford University, Spetember 1991.

[10] Bhatia M. et al. A wavelet-based method for multiscale tomographic reconstruction. *IEEE Trans. on Medical Imaging.* to be published.

[11] Jin Ho Kim et al. the classification of visual stimulus using wavelet transform from eeg signals.
`http://bme.chonbuk.ac.kr/~ kjh91/paper02/Isoes98.htm`.

[12] Mangen J.-M. et al. Méthode itérative de compression d'images radiométriques par la transformée en ondelettes.

[13] Campbell F. and Green D. Optical and retina factors affecting visual resolution. *J. Physiol.*, 181, 1965.

[14] Campbell F. and Kulikowski J. orientation selectivity of the human visual system. *J; Physiol.*, 197, 1966.

[15] Amara Graps. An introduction to wavelets. *IEEE Computational Science and Engineering*, 2(2), 1995.

[16] Maitre H. and Faust B. The nonstationary modelization of images : Statistical properties to be verified. Conf. pattern recogn., May 1978.

[17] Leclercq J.-P. Séminaire d'applications scientifiques, 2000.

[18] Péters J-P. Traitement du signal, 1997-1998. Théorie et Travaux pratiques.

[19] Forinash K. and Lang W. Wavelet analysis of breather mode behavior. *Natural science division*, January 1998.

[20] P.G. Lemarié. Une nouvelles bases d'ondelettes de $\mathbf{L}^2(\mathbb{R})$. *J.Math; Pures et Appl.*, 67, 1988.

[21] Jacques Lewalle. Tutorial on wavelet analysis of experimental data. Syracuse University, April 1995.

[22] Acheroy M. and Mangen J.-M. Progressive wavelet algorithm versus jpeg for the compression of meteosat data. `http://www.rma.ac.be/index.elect.html`.

[23] Wickerhauser M. Lectures on wavelet packet algoritms. Departement of mathematics Washington Universitu, November 1991.

[24] Stéphane Mallat. Multiresolution approximations and wavelet orthonormal bases of $\mathbf{L}^2(\mathbb{R})$. *Trans. Amer. Math. Soc.*, 315, June 1989.

[25] Stéphane Mallat. A theory for multiresolution signal decomposition : The wavelet representation. *IEEE Transactions on Pattern analysis and Machines Intelligence*, II(7) :674–693, July 1989.

[26] Y. Meyer. principe d'incertitude, bases hilbertiennes et algèbres d'opérateurs. *Séminaire Bourbaki*, (662), 1985-1986.

[27] Y. Meyer. Ondelettes et fonctions splines. *Sem. equations aux Dérivées Partielles*, Décember 1986. Ecole Polytechnique de Paris, France.

[28] Colm Mulcahy. Image compression using the haar wavelet transform. *Spelman Science and Math Journal*, pages 22–31. `http://www.spelman.edu/~colm`.

[29] Colm Mulcahy. Plotting and scheming with wavelets. *Mathematics Magazine*, 69(5) :323–343, 1996.

[30] Burt P. and Adelson E. The laplacien pyramid as a compact image code. *IEEE Transactions on Communications*, 31, April 1983.

[31] Lemarié PG. and Meyer Y. Ondelettes et bases hilbertiennes. *Rev. Mat. Iberoamericana*, 2, 1986.

[32] Robi Polikar. The wavelet tutorial.
`http://www.public.iastate.edu/~rpolikar/WAVELETS/`.

[33] Cambridge University Press, editor. 1992.

[34] Buccigrossi R. and Simoncelli E. Image compression via joint statistical characterization in the wavelet domain. *IEEE Transactions on image processing*, 8(12), December 1999.

[35] Gopinath R.A. and Burrus C.S. A trutorial overview of filters banks, wavelets and interrelations. *IEEE Proc. ISCAS - 93.*

[36] Nanda S. and Pearlman W. Tree coding of image subbands. *IEEE transactions on image processing*, I(2) :133–147, April 1992.

[37] A. Said and Pearlman W. Reversible image compression via multiresolution representation and predictive coding. *Proc. SPIE*, 2094 : Visual commun and image processing, November 1993.

[38] A. Said and Pearlman W. A new fast and afficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on circuits and systems for video technology*, 6, June 1996.

[39] A. Said and Pearlman W. An image multiresolution representation for lossless and lossy compression. *IEEE transactions on image processing*, 8, 1999.

[40] Ramchandran K Servetto S. and Orchard M. Image coding based on a morphological representation of wavelet data. *IEE Transactions on image processing*, 1999.

[41] Jérôme Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12) :3445–3462, December 1993.

[42] Carl Taswell. Image compression by parameterized-model coding of wavelet packet near-best bases. `taswell@sccm.stanford.edu`.

[43] Carl Taswell. Speech compression with cosine and wavelet packet near-best bases.

[44] Carl Taswell. Wavbox 4 : A software toolbox for wavelet transforms and adaptative wavelet packet decompositions. *Wavelets and statistics*, december 1994.

[45] Carl Taswell. Specifications and standardsfor reproducibility of wavelet transforms. *Proceedings 7th ICSPAT*, pages 1923–1927, October 1996.

[46] Carl Taswell. Reproducibility standards for wavelet transform algorithms. *Technical Report*, 1998.

[47] Carl Taswell. Wavelet transform compression of functionnal magnetic resonance image sequences. October 1998.

[48] Masami Ueda and Suresh Lodha. Wavelets : an elementary introduction and examples. *UCSC-CRL 94-47*, January 1995.

[49] Neal R. Witten H. and Cleary J.G. *Communications of the ACM*, 30(6), June 1987.

[50] Ramchandran K. Xiong Z. and Orchard M. Space-frequency quantization for wavelet image coding. 1996.

Adresses utiles sur le Web

Beaucoup de mes documents sont issus des pages Web de leurs auteurs et n'ont parfois pas fait l'objet de publication. A toutes fins utiles, je fournis ici une courte revue des principales adresses Internet que j'ai consultés, elle permettra au lecteur de complèter son information le cas échéant.

- `http ://www.amara.com/current/wavelet.html` (sans doute un des meilleurs sites sur la question)
- `http ://www.wavelet.com` (The Wavelet Digest, LA publication virtuelle indispensable)
- `http ://www.mathsoft.com/wavelets.html` (Des articles sur les Ondelettes)
- `http ://www.mat.sbg.ac.at/ uhl/wav.html` (Département de Mathématique de l'université de Salzbourg)
- `http ://cafe.rapidus.net/danilemi/onde.html` (Site en français sur les Ondelettes)
- `http ://liinwww.ira.uka.de/bibliography/Theory/Wavelets` (Moteur de recherche sur les Ondelettes)
- `http ://www.cs.ubc.ca/nest/imager/contributions/bobl/wvlt/top.html` (University of British Columbia Computer Science Dept.)
- `http ://www.rma.ac.be/ jma/wavelet.html`

Annexes

Annexe A

Conventions d'écriture et notions fondamentales

A.1 Notations

- $\mathbb{Z}$ et $\mathbb{R}$ représentent les ensembles des entiers et des réels.
- $\mathbf{L}^2(\mathbb{R})$ est l'espace vectoriel des fonctions de $\mathbb{R}$ à carré intégrable, soit

$$f \in \mathbf{L}^2(\mathbb{R}) \Leftrightarrow \int_{-\infty}^{+\infty} (f(x))^2 \mathrm{d}x \ < \infty$$

  On généralise aisément cette définition pour $\mathbf{L}^2(\mathbb{R}^2)$.
- $\forall f, g \in \mathbf{L}^2(\mathbb{R})$, le *produit scalaire* de f et de g est écrit

$$\langle f(x), g(x) \rangle \ = \ \langle f, g \rangle \ = \ \int_{-\infty}^{+\infty} g(x) f(x) \mathrm{d}x$$

- La norme de f dans $\mathbf{L}^2(\mathbb{R})$ est donnée par

$$\|f\|_{\mathbf{L}^2}^2 \ = \ \|f\|^2 \ = \ \int_{-\infty}^{+\infty} |f(x)|^2 \ \mathrm{d}x$$

- On définit le *produit de convolution* de f et de $g \in \mathbf{L}^2(\mathbb{R})$ par

$$\begin{aligned} (f * g)(x) \ &= (f(u) * g(u))(x) \\ &= \textstyle\int_{-\infty}^{+\infty} f(u) g(x-u) \mathrm{d}u \end{aligned}$$

- Soit $f, g \in \mathbf{L}^2(\mathbb{R}^2)$, le produit scalaire de f et g est donné par

$$\langle f, g \rangle \ = \ \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) \ g(x, y) \ \mathrm{d}x \ \mathrm{d}y$$

- $\mathbf{l}^2(\mathbb{Z})$ est l'espace vectoriel des séquences de réels de carré sommable

$$\mathbf{l}^2(\mathbb{Z}) = \left\{ (\alpha_i)_{i\in\mathbb{Z}} : \sum_{i=-\infty}^{+\infty} |\alpha_i|^2 < \infty \right\}$$

- Soit $(e_i)_{1\leqslant i\leqslant n}$ un ensemble d'éléments d'un espace vectoriel $\mathbb{V}^n$ de dimension n, si $(e_i)_{1\leqslant i\leqslant n}$ est une base orthonormale de $\mathbb{V}^n$, alors

  1.
  $$\|e_i\| = 1, \qquad 1 \leqslant i \leqslant n$$

  2.
  $$\langle e_1, e_j \rangle = \delta_{ij}, \qquad 1 \leqslant i, j \leqslant n$$

  3.
  $$\forall x \in \mathbb{V}^n \ : \ x = \sum_{i=1}^{n} \langle x, e_i \rangle . e_i$$

  L'ensemble des coefficients $\langle x, e_i \rangle$ correspondent à la décomposition de x sur la base (e_i).

A.2 Quelques rappels concernant la Transformée de Fourier

Soit une fonction f, intégrable et T périodique, f peut être décrite par *sa série de Fourier* =

$$f(t) = \sum_{n\in\mathbb{Z}} c_n e^{int}$$

$$\text{avec} \quad c_n = \frac{1}{T} \int_0^T f(t) e^{-i\frac{2\pi}{T}nt} \mathrm{d}t$$

On définit la Transformée de Fourier d'une fonction f par

$$(\mathcal{F}f)(\omega) = \hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-it\omega} f(t) \mathrm{d}t$$

Nous avons que

$$\|\hat{f}\|_{\mathbf{L}^2} = \|f\|_{\mathbf{L}^2}$$

$$|\hat{f}(\omega)| \leqslant (2\pi)^{-\frac{1}{2}} \|f\|_{\mathbf{L}^1} = (2\pi)^{-\frac{1}{2}} \int_{-\infty}^{+\infty} |f(t)| \mathrm{d}t$$

$$\mathcal{F}\left(\frac{\mathrm{d}^l}{\mathrm{d}x^l} f\right)(\omega) = (i\omega)^l (\mathcal{F}f)(\omega)$$

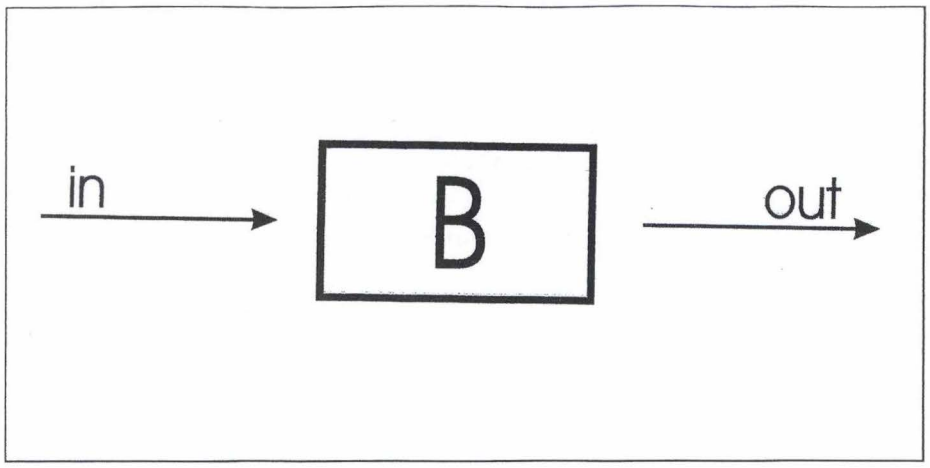


FIG. A.1 – *Un système de traitement du signal quelconque*

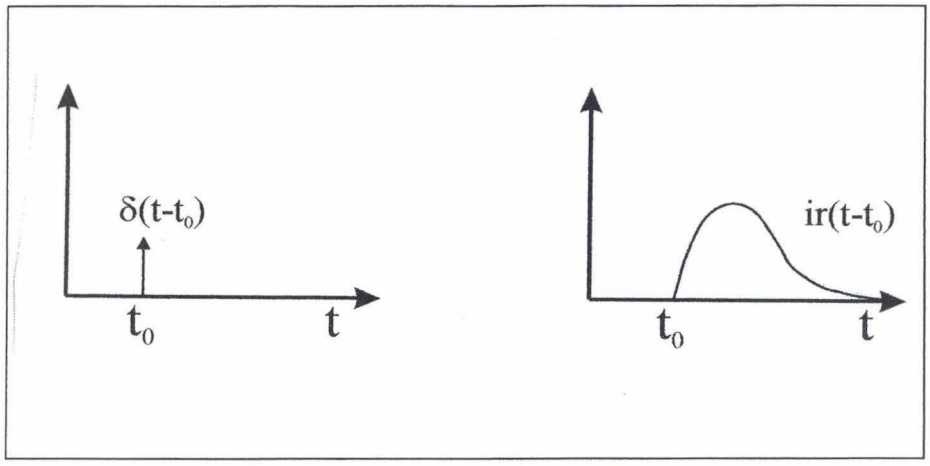


FIG. A.2 – *Un signal impulsionnel unité et la réponse du système*

A.3 Notions de traitement du signal : réponse impulsionnelle, produit de convolution et théorème de Shannon

Imaginons le système suivant (figure A.1). Nous avons une boîte noire B qui prend en entrée un signal $in(t)$ et dont la sortie est un autre signal $out(t)$. On caractérise B par sa réponse impulsionelle. Il s'agit de la réponse de B à un signal composé d'une impulsion d'amplitude unitaire et de largeur nulle (il s'agit d'une fonction delta de Dirac). Pour un système physique, on peut imaginer la réponse impulsionelle $ir(t)$ comme illustré sur la figure A.2 : la réponse à une excitation.

Tout signal d'entrée $in(t)$ peut être vu comme une suite d'impulsions dont les

amplitudes sont données par $in(t)$. B répond toujours de la même manière à ces impulsions, par $r(t)$, mais plus ou moins fortement en fonction de l'amplitude de l'impulsion. Donc, $out(t)$ est composé de la réponse à l'impulsion $e(t)$ mais aussi des réponses aux impulsions précédentes et qui ne sont pas encore terminées (il faut un certain temps pour que B revienne à l'équilibre après une excitation). Ainsi, on écrit :

$$out(t) = \int_{-\infty}^{+\infty} in(\tau) ir(t-\tau) \mathrm{d}t$$

soit

$$out(t) = (in(\tau) * ir(\tau))(t) \tag{A.1}$$

Donc, si nous connaissons $ir(t)$ pour le système B, nous pouvons calculer la réponse de B à tout signal d'entrée par le produit de convolution de ce signal avec la réponse impulsionnelle. Examinons ce qui se passe au niveau fréquentiel en prenant la Transformée de Fourier de $out(t)$.

$$\widehat{out}(\omega) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} in(\tau) ir(t-\tau) e^{-i\omega t} \, \mathrm{d}\tau \mathrm{d}t$$

En posant $t - \tau = t'$, on arrive à

$$\widehat{out}(\omega) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} in(\tau) ir(t') e^{-i\omega(t'+\tau)} \, \mathrm{d}\tau \mathrm{d}t'$$

On peut séparer cette intégrale en deux :

$$\widehat{out}(\omega) = \int_{-\infty}^{+\infty} in(\tau) \, e^{-i\omega\tau} \, \mathrm{d}\tau \int_{-\infty}^{+\infty} ir(t') \, e^{-i\omega t'} \, \mathrm{d}t'$$

Et donc

$$\widehat{out}(\omega) = \widehat{in}(\omega)\widehat{ir}(\omega) \tag{A.2}$$

Inversément, nous pourrions, à partir de A.2, retrouver A.1. En fait, le produit de convolution et la Transformée de Fourier sont liés par deux relations fondamentales :

$$\begin{aligned} h(t) &= (f * g)(t) \Leftrightarrow h(\omega) = f(\omega)g(\omega) \\ h(\omega) &= (f * g)(\omega) \Leftrightarrow h(t) = f(t)g(t) \end{aligned}$$

Donc, une convolution en temps se traduit par une multiplication en fréquence (et inversément), une convolution en fréquence se traduit par une multiplication en temps (et inversément).

Nous terminerons cette annexe par un rappel du théorème de Shannon qui est réellement fondamental en traitement du signal. Soit $f \in \mathbf{L}^2(\mathbb{R})$, une fonction dont la

Transformée de Fourier est à support compact (c'est-à-dire $\hat{f}(\omega) = 0$ pour $|\omega| > \Omega$). Supposons que $\Omega = \pi$ pour simplifier. On peut représenter $\hat{f}$ par sa série de Fourier :

$$\hat{f}(\omega) = \sum_{n\in\mathbb{Z}} c_n e^{-in\omega}$$

On a donc

$$\begin{aligned} c_n &= \tfrac{1}{2\pi}\int_{-\pi}^{+\pi} e^{in\omega}\hat{f}(\omega)\,\mathrm{d}\omega \\ &= \tfrac{1}{2\pi}\int_{-\infty}^{+\infty} e^{in\omega}\hat{f}(\omega)\,\mathrm{d}\omega \;=\; \tfrac{1}{\sqrt{2\pi}} f(n) \end{aligned}$$

On peut donc écrire

$$\begin{aligned} f(x) &= \tfrac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{ix\omega}\hat{f}(\omega)\,\mathrm{d}\omega \\ &= \tfrac{1}{\sqrt{2\pi}}\int_{-\pi}^{+\pi} e^{ix\omega}\textstyle\sum_{n\in\mathbb{Z}} c_n e^{-in\omega}\,\mathrm{d}\omega \\ &= \tfrac{1}{\sqrt{2\pi}}\textstyle\sum_{n\in\mathbb{Z}} c_n \int_{-\pi}^{+\pi} e^{i(x-n)\omega}\,\mathrm{d}\omega \\ &= \tfrac{1}{\sqrt{2\pi}}\textstyle\sum_{n\in\mathbb{Z}} c_n \frac{1}{i(x-n)}\left(e^{i\pi(x-n)} - e^{-i\pi(x-n)}\right) \\ &= \tfrac{1}{\sqrt{2\pi}}\textstyle\sum_{n\in\mathbb{Z}} c_n \frac{2\sin\pi(x-n)}{x-n} \\ &= \textstyle\sum_{n\in\mathbb{Z}} f(n)\,\frac{\sin\pi(x-n)}{\pi(x-n)} \end{aligned}$$

Donc, cette dernière ligne nous indique que f est entièrement déterminée par ses valeurs échantillonnées : $f(n)$. Nous pouvons même réécrire cette formule pour $\Omega \neq \pi$, on a :

$$f(x) = \sum_{n\in\mathbb{Z}} f(n\frac{\pi}{\Omega})\frac{\sin(\Omega x - n\pi)}{\Omega x - n\pi}$$

Cette relation constitue le théorème de Shannon. ■

Ainsi, nous voyons que f est totalement déterminé par ses valeurs échantillonnées à la fréquence $2\nu_{max}$ où $2\pi\nu_{max} = \Omega$ (auquel cas $2\nu_{max} = \frac{\Omega}{\pi}$). Cette fréquence est appellée fréquence de Nyquist. Elle est extrêmement importante. En effet, si on échantillonne un signal à une fréquence supérieure à $2\nu_{max}$ (c'est-à-dire deux fois la fréquence maximale du signal échantillonné), on pourra le reconstruire facilement, en utilisant des fonctions à décroissance plus rapide que $\frac{\sin(x)}{x}$. Par contre, si on échantillonne un signal avec une fréquence inférieure à $2\nu_{max}$, il sera impossible de reconstruire le signal correctement. Ce phénomène, très connu en traitement du signal, est appellé aliasing. Il se manifeste en acoustique par des parasites audibles ressemblant à des claquements métalliques.

Annexe B

Fonction de Battle-Lemarié

Dans cette annexe, nous décrivons l'Ondelette de Battle-Lemarié. Cette fonction génère une Approximation Multirésolution de $\mathbf{L}^2(\mathbb{R})$.

L'espace vectoriel $\mathbb{V}_1$ est l'espace de toutes les fonctions de $\mathbf{L}^2(\mathbb{R})$ qui sont p fois continuemment différentiables et égales à un polynôme d'ordre $2p+1$ sur les intervalles $[k, k+1], k \in \mathbb{Z}$.

On peut montrer que la Transformée de Fourier de la fonction d'échelle associée est

$$\hat{\phi}(\omega) = \frac{1}{\omega^n \sqrt{\sigma_{2n}(\omega)}} \text{ où } n = 2 + 2p$$

avec la fonction

$$\sigma_{2n}(\omega) = \sum_{k=-\infty}^{+\infty} \frac{1}{(\omega + 2k\pi)^n}$$

Le théorème 2 nous dit que $\hat{\phi}(2\omega) = H(\omega)\hat{\phi}(\omega)$. On arrive donc à l'expression de $H(\omega)$:

$$\mathbf{H}(\omega) = \sqrt{\frac{\sigma_{2n}(\omega)}{2^{2n}\sigma_{2n}(2\omega)}}$$

Par le théorème 3, nous pouvons avoir l'expression de la Transformée de Fourier de l'Ondelette associée :

$$\hat{\psi}(\omega) = e^{-i(\frac{\omega}{2})} \overline{H\left(\frac{\omega}{2} + \pi\right)} \hat{\phi}\left(\frac{\omega}{2}\right)$$

$$= \frac{e^{-i(\frac{\omega}{2})}}{\omega^n} \frac{\sqrt{\sigma_{2n}(\frac{\omega}{2}+\pi)}}{\sqrt{\sigma_{2n}(\omega)\sigma_{2n}(\frac{\omega}{2})}}$$

La fonction $\psi(\omega)$ est exponentiellement décroissante.

Le tableau B.1 donne les coefficients $h(n)_{0 \leqslant n \leqslant 11}$. Ce filtre est symétrique. On peut obtenir $g(n)$ via la relation

$$g(n) = (-1)^{1-n} h(1-n)$$

n	$h(n)$	n	$h(n)$
0	0.542	6	0.012
1	0.307	7	-0.013
2	-0.035	8	0.006
3	-0.078	9	0.006
4	0.023	10	-0.003
5	-0.030	11	-0.002

TAB. B.1 – Coefficient $h(n)$

Annexe C

Quelques applications de la Transformée en Ondelettes

Cette annexe est consacrée à montrer combien les applications de la Transformée de Fourier sont nombreuses et diversifiées. Il ne s'agit pas d'une présentation détaillée de celles-ci et je n'ai pas la prétention d'être exhaustif. Nous opèrerons un survol rapide de quelques applications importantes sur lesquelles j'ai pu trouver des informations. Une partie de cette annexe est tirée de [15].

1. Compression d'images

   Je ne vais pas refaire ici tout ce mémoire mais simplement mentionner l'utilisation qui a été faite de la compression via les Ondelettes au FBI. Depuis 1924 à nos jours, le US Federal Bureau of Investigation a collecté près de 30 millions d'empreintes, la plupart sur support papier, à l'ancienne. Le problème s'est rapidement posé de la distribution de ces empreintes dans les différentes agences du pays. Les copies papier étant souvent de mauvaise qualité, il a été décidé d'opérer par voie informatique et le FBI a établi un standard pour la digitalisation et la compression des images d'empreintes. Ce standard est basé sur la Transformée en Ondelettes Discrète.

   Pour la petite histoire, on peut mentionner que les empreintes sont stockées à une résolution de 500 pixels par pouce, en 256 niveaux de gris. Une seule empreinte contient 700 000 pixels et requiert donc 0.6 Mbytes pour être stockée sans compression. Une paire de mains demande donc 6 Mbytes et l'ensemble des archives du FBI 200 terabytes. Lorsque ce standard a été développé, les gestionnaires ont calculé un coût de stockage non-compressé de 200 millions de dollars. On comprend que la compression les ait interessé !

   Actuellement, des algorithmes spécifiques de compression sont élaborés en Belgique pour traiter des images issues du satellite Meteosat [22, 12].

2. Transmission d'images

   La Représentation en Ondelettes d'une image est une découpe hiérarchique de l'information de celle-ci en bande de fréquences (cfr chapitres 2 et 5). La

FIG. C.1 – *Nelson Mandela transmis par les ondelettes. L'image est reconstruite avec* $0,5\%, 1\%, 2\%, 4\%, 10\%$ *et* 50% *des coefficients*

transmission des images a été très rapidement investie par les Ondelettes. En effet, pour transmettre de façon simple une image en faisant en sorte que le récepteur puisse rapidement voir de quoi il s'agit, il suffit de transmettre les coefficients de la Représentation en Ondelettes par bandes en commençant par les bandes de basses résolutions. Le récepteur peut alors reconstruire les approximations de l'image aux résolutions concernées. Ce système est déjà implémenté sur Internet et une société vend un plugin pour Netscape qui utilise cette technique (`http://www.summus.com`). La figure C.1 montre les reconstructions successives effectuées en cours de transmission pour une image de Nelson Mandela ([28, 34]).

3. Suppression de bruit

   La suppression de bruit est liée aux mêmes techniques que la compression d'image. Comparer les coefficients à une valeur de seuil et les supprimer s'ils sont inférieurs en valeur absolue consiste à supprimer les hautes fréquences de faibles amplitudes (les détails non-significatifs). Pour certains signaux, ces fréquences correspondent aux bruits indésirables. Ainsi, la figure C.2 montre un signal RMN avant et après traitement ([15, 47]). La méthode est simple :
   – effectuer une Transformée en Ondelettes discrète

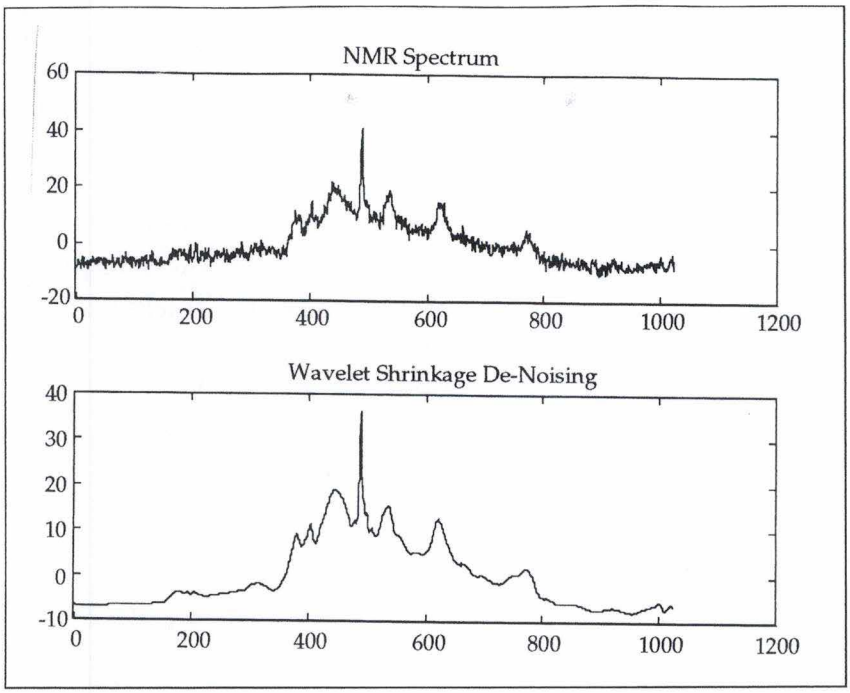


FIG. C.2 – *Signal RMN avant et après traitement pour la suppression du bruit*

- filtrer les coefficients par rapport à un seuil
- effectuer une Transformée en Ondelettes Discrète inverse

4. Astronomie

   En astronomie, les propriétés très particulières de la Transformée en Ondelettes ont permis de faire de nouvelles découvertes. En effet, les traitements habituels des spectres de certaines étoiles ne donnaient pas de résultat et la Transformée en Ondelettes a permis de tirer des conclusions en montrant que les variations de ces spectres étaient les mêmes à toutes les résolutions.

5. Génération de sons

   Les Ondelettes sont utilisées en musique pour générer de nouveaux sons et imiter les instruments. Il s'agit simplement d'approximer les spectres des différents instruments via une Transformée en Ondelettes. On stocke alors les coefficients et, lorsqu'on veut rejouer un son, il suffit de faire une Transformée en Ondelettes inverse et jouer le son voulu. Les Ondelettes s'appliquent aux sons comme aux autres signaux.

6. Géophysique, météorologie et médecine

   L'analyse par Transformée en Ondelettes en géophysique et en météorologie est abondamment étudiée [7]. Il semblerait que cette technique s'adapte particulièrement bien à l'étude des signaux quasi chaotiques que sont, par exemple, les données géophysiques ou les électro-encéphalogrammes. C'est tou-

jours cette capacité à analyser les signaux à plusieurs résolutions différentes avec les mêmes moyens qui semble être à l'origine de cet attrait. Dans le cas de la médecine, la Transformée en Ondelettes sert à classifier les électro-encéphalogrammes. Une approche empirique a permis de définir des spectres types auxquels on rapporte un signal à classer. Par comparaison des propriétés aux différentes résolutions (6 niveaux), on parvient à opérer un classement permettant d'aider au diagnostic.

En médecine, on utilise également la Transformée en Ondelettes pour reconstruire le volume en trois dimensions d'objets dont on a pris des images à différents angles. Dans ce cas précis, il s'agit d'images issues de tomographes [10].

Annexe D

Courte excursion dans la compression de sons

J'ai été amené à m'intéresser presque par hasard à la compression de sons. En effet, ce domaine ne rentre pas dans le cadre de ce travail. Cependant, je pensais qu'il devait en aller des sons comme des images. Disposant déjà d'une toolkit permettant d'appliquer la Transformée en Ondelettes, le travail à réaliser pour appliquer cette technique aux sons était assez restreint (tout au moins en apparence). J'ai donc développé deux classes JAVA permettant de manipuler les fichiers .wav, format standard des sons, et de leur appliquer une Transformée en Ondelettes. Comme il est plus délicat d'imposer à un fichier son d'avoir un nombre d'échantillons égal à un multiple de deux, la Transformée en Ondelettes est appliquée par blocs de longueur égale à des multiples de deux. Je n'ai pas cherché à aller plus loin dans le traitement.

Les résultats sont assez surprenants. En effet, un fichier .wav reconstruit à partir de sa Transformée en Ondelettes Discrète avec 100% des coefficients est absolument parfait : on n'entend aucune différence par rapport au fichier original. Cependant, dès que l'on supprime quelques pourcents des coefficients, il apparaît énormément de bruit dans le fichier reconstruit, au point de rendre le signal inaudible.

Je m'interroge toujours sur ce phénomène. Je n'ai pas poussé plus loin étant donné que ceci sortait du cadre de mon travail mais j'ai trouvé certaines références qui indiquent que la Transformée en Ondelettes est déjà utilisée dans ce domaine. Je pense donc que ce sujet pourrait faire l'objet d'une étude plus poussée. Au vu de mes dernières recherches sur le web, il semble même que ce soit un sujet prometteur.

Annexe E

Programmes réalisés

E.1 Toolkit JAVA : le package WaveletTools

E.1.1 La classe BitmapHandler

```
 package WaveletTools;

import java.io.*;
import java.util.Vector;

class BitmapHandler
{
    /* This class provide a toolkit to handle bmp image (windows format).
    At this time only one encoding scheme is supported : 8 bits per pixel and no compression (I hope I'll soon be able to program other
    encoding scheme -6 in total-)

    */

    // bitmap format parameters

    // 1) header (14 bytes)

    private int[] signature = {66, 77};             // BM in iso latin ascii chart
    private int[] fileSize = new int[4];             // 4 bytes
    private int[] unused = {0, 0, 0, 0};
    private int[] dataOffset = new int[4];             // 4 bytes

    // 2) info header (40 bytes)

    private int[] infoHeaderSize = {40, 0, 0, 0};        // 40
    private int[] width = new int[4];            // width of image (in pixels)
    private int[] heigth = new int[4];            // heigth of image (in pixels)
    private int[] planes = {1, 0};                // number of planes (1)
    private int[] bitCount = new int[2];            // bits per pixel (1 = monochrome; 4 = 16 colors; 8 = 256 colors; 16 = 65536 colors;
                            // 24 = 16M colors)
    private int[] compression = new int[4];            // compression scheme (0 = no compression; 1 = 8bit RLE encoding; 2 = 4bit RLE encoding)
    private int[] imageSize = new int [4];            // size of image in bytes; can be = 0 if compression = 0
    private int[] Xresolution = new int[4];            // horizontal resolution (pixels/meter)
    private int[] Yresolution = new int[4];            // vertical resolution (pixels/meter)
    private int[] colorUsed = new int[4];            // number of colors used
    private int[] colorImportant = new int[4];        // number of important colors (0 = all)

    private int totHeadSize = 54;                // total length of headers : 54 bytes

    // 3) color table

    private int[] colorTable;                // color table is only present if bitCount <= 8
                            // structure :    1 byte red intensity
                            //        1 byte green intetnsity
                            //        1 byte blue intensity
                            //        1 byte unused (= 0)
                            // the table has a dimension = 'colorUsed'*4

    private int greyLevelColorTableSize = 1024;        // standart size of a greyLevels Color Table or any 8 bpp color Table

    /* The pixels are stored bottom-up and left-to-right. Pixel lines are padded with zero to end on 32bit boundary. Color indices are 0
    based (0 represent the first color table entry and 255 the 256th entry).
```

```
*/

private Vector param = null;                // this vector must contains all parameter above \huge {IN GOOD ORDER!!!}
                            // it is fill in standart constructor -> this constructor must be update in case of format changes

// parameters functions

public BitmapHandler()
{
    // constructor only fill param with all parameter for re-use later
    // beware to update this method if some format changes are made...

    param = new Vector();

    param.add(signature);
    param.add(fileSize);
    param.add(unused);
    param.add(dataOffset);
    param.add(infoHeaderSize);
    param.add(width);
    param.add(heigth);
    param.add(planes);
    param.add(bitCount);
    param.add(compression);
    param.add(imageSize);
    param.add(Xresolution);
    param.add(Yresolution);
    param.add(colorUsed);
    param.add(colorImportant);
}

public void setFileParameters(String file)
{
    // set all the current parameters with those of file (it should be a .bmp)

    try
    {
        FileInputStream bmp = new FileInputStream(file);

        int[] parameter = null;
        for (int i = 0; i < param.size(); i++)
        {
            parameter = (int[]) param.get(i);
            for (int j = 0; j < parameter.length; j++)
            {
                parameter[j] = bmp.read();
            }
        }
    }
    catch (IOException e)
    {
        System.out.println("IOException occured : " + e.getMessage());
    }

}
```

```
public void printFileParameters(String file)
{
    // print all the current parameters of file (it should be a .bmp)

    try
    {
        FileInputStream bmp = new FileInputStream(file);

        int[] parameter = null;
        for (int i = 0; i < param.size(); i++)
        {
            parameter = (int[]) param.get(i);
            for (int j = 0; j < parameter.length; j++)
            {
                parameter[j] = bmp.read();
                System.out.println(parameter[j]);
            }
        }
    }
    catch (IOException e)
    {
        System.out.println("IOException occured : " + e.getMessage());
    }

}

public void generateGreyLevelColorTable()
{
    // fill colorTable with a greyLevels color Table
    // Such a table has structure : 0000, 1110, 2220, 3330, ...

    colorTable = new int[greyLevelColorTableSize];

    for (int i = 0; i < 256; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            colorTable[(i * 4) + j] =  i;
        }
        colorTable[(i * 4) + 3] = 0;
    }
}

// working attributes

String fileName = "";
int[] rasterData = null;

// handling functions

public int[] getRasterData()
{
    return rasterData;
}

public void setRasterData(int[] data)
```

```
{
    // set rasterData values to those of data

    rasterData = new int[data.length];
    for (int i= 0; i < data.length; i++)
    {
        rasterData[i] = data[i];
    }
}

public int[] getSize()
{
    int widthB = width[0] + (width[1] * 256) + (width[2] * 65536) + (width[3] * 16777216);
    int heigthB = heigth[0] + (heigth[1] * 256) + (heigth[2] * 65536) + (heigth[3] * 16777216);
    int[] out = {widthB, heigthB};
    return out;
}

public void extractBMPData(String file)
{
    // extract data fom bmp file to rasterData (!without padding!)
    // at this timle only on encoding scheme is supported : 8 bpp, no compression
    // put file Name to fileName

    fileName = file;
    setFileParameters(file);

    int widthB = width[0] + (width[1] * 256) + (width[2] * 65536) + (width[3] * 16777216);
    int heigthB = heigth[0] + (heigth[1] * 256) + (heigth[2] * 65536) + (heigth[3] * 16777216);
    int imageSizeB = (imageSize[0] + (imageSize[1] * 256) + (imageSize[2] * 65536) + (imageSize[3] * 16777216));
    int line = widthB - (widthB % 4);
    int data = 0;

    rasterData = new int[imageSizeB - (heigthB * (widthB % 4))];

    //System.out.println("width: " + widthB + ", heigth: " + heigthB + ", imageSize: " + imageSizeB + ", line : " + line + ", #data: " + rasterData.length + ", padding: " +

    if ((bitCount[0] == 8) && (compression[0] == 0))
    {
        try
        {
            FileInputStream bmp = new FileInputStream(file);

            // goto Raster data

            for (int i = 0; i < (greyLevelColorTableSize + totHeadSize); i++)
            {
                bmp.read();
            }

            // read Raster data

            for (int i = 0; i < heigthB; i++)
            {
                for (int j = 0; j < line; j++)
                {
```

```
                    data = bmp.read();
                    rasterData[j + (i * line)] = data;
                }
                for (int k = 0; k < (widthB % 4); k++)           // padding -> not read...
                {
                    bmp.read();
                }
            }

            bmp.close();
        }
        catch (IOException e)
        {
            System.out.println("IOException occured : " + e.getMessage());
        }
    }
    else
        System.out.println("Unsupported Format...");

}

public void composeBMP(String file)
{
    // compose a file named file.bmp in bmp format with rasterData and current parameters (grey levels assumed)
    // at this time only on encoding scheme is supported : 8 bpp, no compression
    // put file.bmp to fileName

    fileName = file + ".bmp";
    generateGreyLevelColorTable();
    int widthB = width[0] + (width[1] * 256) + (width[2] * 65536) + (width[3] * 16777216);
    int heigthB = heigth[0] + (heigth[1] * 256) + (heigth[2] * 65536) + (heigth[3] * 16777216);
    int imageSizeB = (imageSize[0] + (imageSize[1] * 256) + (imageSize[2] * 65536) + (imageSize[3] * 16777216));
    int line = widthB - (widthB % 4);
    int data = 0;

    //System.out.println("width: " + widthB + ", heigth: " + heigthB + ", imageSize: " + imageSizeB + ", line : " + line + ", #data: " + rasterData.length + ", padding: " +

    if ((bitCount[0] == 8) && (compression[0] == 0))
    {
        try
        {
            FileOutputStream bmp = new FileOutputStream(file + ".bmp");

            // print header

            int[] parameter = null;
            for (int i = 0; i < param.size(); i++)
            {
                parameter = (int[]) param.get(i);
                for (int j = 0; j < parameter.length; j++)
                {
                    bmp.write(parameter[j]);
                }
            }

            // print color table
```

```
                for (int j = 0; j < colorTable.length; j++)
                {
                    bmp.write(colorTable[j]);
                }

                // print Raster data

                for (int i = 0; i < heigthB; i++)
                {
                    for (int j = 0; j < line; j++)
                    {
                        bmp.write(rasterData[j + (i * line)]);
                    }
                    for (int k = 0; k < (widthB % 4); k++)            // padding with 0 ;-)
                    {
                        bmp.write(0);
                    }
                }

                bmp.close();
            }
            catch (IOException e)
            {
                System.out.println("IOException occured : " + e.getMessage());
            }
        }
        else
            System.out.println("Unsupported Format...");


    }
}
```


E.1.2 La classe WTTransformer

```
package WaveletTools;

import java.util.Vector;

/*
    WTTransformer provide a tool to apply wavelet transform in n dimensions with one of the supported wavelets family. I try to make it
    fully configurable and easy to use.  I advice to use the WTTransformer contructor with all parameters instead of each particular methods
    (exept if you know what you'r doin').

    To use WTTransformer, create it with the standart constructor and call wtn().  The result is then in the array called out and you can take it back
    with function getOut().  Beware that out have same format as in.

    WTTransformer is partialy a translation in java of functions from "Numerical Recipies in Fortran77" (Cambridge University Press 1992).

    parameter comments:
        ndim :          the tranformer perform an ndim-dimensional WT.
        nn :          the size of each dimension of input data (i.e. the real number of values in each dimension); dimension of nn is ndim
```

```
            and each component MUST all be powers of 2!
        in :          input array; This array have a $ size = \prod_{i=1}^ndim nn[i] $ (\LaTeX format ;-) and the data are store in
            respect to invert order of nn (as arrays are stored in FORTRAN). That means stored by columns for a two dimensionnal
            array.
        out :          output data, stored as input array in.
        cc :          array that contains the coefficients for the wavelet family choosed by user (initialized by pwtSet). cc is
            the smoothing filter.
        cr :          cr is the quadrature mirror filter of cc needed by WT algorithm.
        ioff and joff:      index of the centering values of cc and cr
        isign :      if isign=1 wtn perform direct wavelet transform, if isign=-1 wtn perform inverse wavelet transform
        motherWT :      the conventionnal name of the mother wavelet user wants to use.
            The supported mother wavelets are (at this time):
                Daubechies :      DAUB (4, 12, 20)
                The Haar :         HAAR
        ncoef :      the number of coefficients the user wants to use (ncoef = 4, 12 or 20 for Daubechies wavelets).
        data :          data is an instance of DataWavelet. This class contains all the coefficients for all the families of wavelets.
        mode :          mode determines if user wich that WTTransformer terminate all if an unrecoverable error occured (1=yes that's
            the default case, -1=no)

        working :      array used for computation

    In practice, nn, in, cc and cr have all a dimension = (real_dimension + 1) with there first element equal to 0 for facility reasons.
    User may ignore this fact.

*/

class WTTransformer
{
    private double[] in = null, out = null, cc = null, cr = null, working = null;
    private int[] nn = null;
    private int ndim = 0, isign = 0, ncoef = 0, mode=1, ioff = 0, joff = 0;
    private String motherWT = "";
    private DataWavelet data;

    public WTTransformer(double[] inInit, int[] nnInit, int ndimInit, int isignInit, String motherWTInit, int ncoefInit, int modeInit)
    {

        // create a WTTransformer with initialized values of in, nn, isign, motherWT, ncoef and cc/cr

        super();
        dataSet(new DataWavelet());
        inSet(inInit);
        workingInit();
        nnSet(nnInit);
        ndimSet(ndimInit);
        isignSet(isignInit);
        motherWTSet(motherWTInit);
        ncoefSet(ncoefInit);
        modeSet(modeInit);
        pwtSet();
        outInit(in.length - 1);


    }

    public void reinitialize(double[] inInit, int[] nnInit, int ndimInit, int isignInit, String motherWTInit, int ncoefInit, int modeInit)
```

```
{
    inSet(inInit);
    workingInit();
    nnSet(nnInit);
    ndimSet(ndimInit);
    isignSet(isignInit);
    motherWTSet(motherWTInit);
    ncoefSet(ncoefInit);
    modeSet(modeInit);
    pwtSet();
    outInit(in.length - 1);
}

public void pwtSet()
{

    // set cc and cr values to those of the mother wavelet specified by motherWT and ncoef; set values of ioff and joff

    double[] coef = data.getData(motherWT, ncoef);

    if (coef == null)
    {
        System.out.println("Unrecoverable error with the wavelet's coefficients");
        if (mode == 1)
        {
            System.exit(0);
        }
    }
    else
    {
        int[] center = data.getCenter(motherWT, ncoef);        // no check on center value because it's already done...
        ioff = center[0];
        joff = center[1];

        if (motherWT.equals("LBSP") || motherWT.equals("BURT"))
        {
            ncoef = coef.length / 2;

            cc = new double[ncoef + 1];
            cr = new double[ncoef + 1];
            cc[0] = 0;
            cr[0] = 0;

            for (int i = 1; i < cc.length; i++)
            {
                cc[i] = coef[i - 1];
                cr[i] = coef[ncoef + i - 1];
            }

        }
        else
        {
            ncoef = coef.length;

            cc = new double[coef.length + 1];
            cr = new double[coef.length + 1];
```

```
            cc[0] = 0;
            cr[0] = 0;

            int sig = -1;
            for (int i = 1; i < cc.length; i++)
            {
                cc[i] = coef[i - 1];
                cr[ncoef + 1 - i] = sig * cc[i];
                sig = -sig;
            }
        }
    }

}

private void pwt(double[] work, int n)
{

    /* partial WT: applies an arbitrary wavelet filter to data vector work(1:n) (for isign=1) or applies its transpose (for
    isign=-1). Used hierarchically by routine wtn. The actual filter is determined by preceding (and required) call to pwtset.
    */

    double[] wksp = new double[work.length];
    int i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;
    double ai,ai1;

    if (n < 4) return;

        nmod = ncoef * n;                          // a positive constant equal to zero mod n
        n1 = n - 1;                               // mask of all bits, since n a power of 2 (see below)
        nh = n / 2;
    for (j = 1; j <= n; j++)
    {
        wksp[j] = 0;
    }

    if (isign >= 0)                               // apply filter
    {
        ii = 1;
        for (i = 1; i <= n; i += 2)
        {
            ni = i + nmod + ioff;                  // "pointer" to be incremented and wrap-around
            nj = i + nmod + joff;
            for (k = 1; k <= ncoef; k++)
            {
                jf = n1 & (ni+k);                 // use bitwise and to wrap-around the "pointers"
                jr = n1 & (nj+k);
                wksp[ii] = wksp[ii] + cc[k] * work[jf+1];
                wksp[ii+nh] = wksp[ii+nh] + cr[k] * work[jr+1];
            }
            ii++;
        }
    }
    else                                          // apply transpose filter
    {
        ii = 1;
```

```
            for (i = 1; i <= n; i += 2)
            {
                ai = work[ii];
                ai1 = work[ii+nh];
                ni = i + nmod + ioff;
                nj = i + nmod + joff;
                for (k = 1; k <= ncoef; k++)
                {
                    jf = (n1 & (ni+k)) + 1;
                    jr = (n1 & (nj+k)) + 1;
                    wksp[jf] = wksp[jf] + cc[k] * ai;
                    wksp[jr] = wksp[jr] + cr[k] * ai1;
                }
                ii++;
            }
        }

        for (j = 1; j <= n; j++)
        {
            work[j] = wksp[j];                          // copy the results back from workspace
        }

        return;

    }

    public void wtn()
    {

        /* Replace working by its ndim-dimensional DWT (isign=1) or by its invert DWT (isign=-1) and copy the results to out.
        wtn uses pwt to perform the hierachical algorithm of multi-dimensional DWT.
        */

        int i1,i2,i3,idim,k,n,nnew,nprev,nt,ntot;
        double[] wksp = new double[working.length];

        ntot = 1;
        for (idim = 1; idim <= ndim; idim++)
        {
            ntot = ntot * nn[idim];
        }
        nprev = 1;
        for (idim = 1; idim <= ndim; idim++)                    // main loop over the dimensions
        {
            n = nn[idim];
            nnew = n * nprev;
            if (n > 4)
            {
                for (i2 = 0; i2 <= (ntot-1); i2 += nnew)
                {
                    for (i1 = 1; i1 <= nprev; i1++)
                    {
                        i3 = i1 + i2;
                        for (k = 1; k <= n; k++)
                        {
                            wksp[k] = working[i3];          // copy the revelant row or column or etc. into workspace
```

```
                    i3 += nprev;
                }
                if (isign >= 0)                // do one-dimensional wavelet transform
                {
                    nt = n;
                    while (nt >= 4)
                    {
                        pwt(wksp, nt);
                        nt /= 2;
                    }
                }
                else                    // or inverse transform
                {
                    nt = 4;
                    while (nt <= n)
                    {
                        pwt(wksp, nt);
                        nt *= 2;
                    }
                }
                i3 = i1 + i2;
                for (k = 1; k <= n; k++)        // copy back from workspace
                {
                    working[i3]=wksp[k];
                    i3 += nprev;
                }
            }
        }
    }
    nprev = nnew;
}
outSet();                               // fill out
return;

}

public void inSet(double[] inInit)
{

    // fill in with values of inInit

    in = new double[inInit.length + 1];
    in[0] = 0;
    for (int i = 1; i < in.length; i++)
        in[i] = inInit[i - 1];

}

public void nnSet(int[] nnInit)
{

    // fill nn with values of nnInit

    nn = new int[nnInit.length + 1];
    nn[0] = 0;
    for (int i = 1; i < nn.length; i++)
```

```
        nn[i] = nnInit[i - 1];
    }

    public void ndimSet(int ndimInit)
    {

        // set value of ndim at ndimInit

        ndim = ndimInit;

    }

    public void isignSet(int isignInit)
    {

        // set value of isign at isignInit

        isign = isignInit;

    }

    public void motherWTSet(String motherWTInit)
    {

        // set value of motherWT at motherWTInit

        motherWT = motherWTInit;
    }

    public void ncoefSet(int ncoefInit)
    {

        // set value of ncoef to ncoefInit

        ncoef = ncoefInit;

    }

    public void modeSet(int modeInit)
    {

        // set mode value at modeInit

        mode = modeInit;

    }

    public void dataSet(DataWavelet d)
    {

        // set data to d

        data = d;
    }
```

```
public void outInit(int inDim)
{
    // initialize out with dimension inDim. WARNING: inDim must be the dimension of in!!!

    out = new double[inDim];
}

public void workingInit()
{

    // fill working with values of in

    working = new double[in.length];

    for (int i = 0; i < in.length; i++)
        working[i] = in[i];

}

private void outSet()
{
    // copy the results from working to out

    for (int i = 1; i < working.length; i++)
        out[i - 1] = working[i];

}

public Vector getData()
{
    // to have information about the WTTransformer
    // return a vector with format : [motherWT; ncoef; isign; ndim; nn]
    // WARNING : ncoef, isign and ndim are Integer instead of int!

    Vector output = new Vector();
    Integer isignI = new Integer(isign);
    Integer ndimI = new Integer(ndim);
    Integer ncoefI = new Integer(ncoef);

    output.add(motherWT);
    output.add(ncoefI);
    output.add(isignI);
    output.add(ndimI);
    output.add(nn);

    return output;
}

public double[] getCc()
{

    // return cc array without its first element (=0)

    double[] ccOut = new double[cc.length - 1];

    for (int i = 1; i < cc.length; i++)
```

```
            ccOut[i - 1] = cc[i];

        return ccOut;
    }


    public double[] getIn()
    {

        // return in array without its first element (=0)

        double[] inOut = new double[in.length - 1];

        for (int i = 1; i < in.length; i++)
            inOut[i - 1] = in[i];

        return inOut;
    }


    public double[] getOut()
    {

        // return out array

        return out;
    }


}
```


E.1.3 La classe DataCompressor

```
package WaveletTools;

import java.util.Vector;
import java.math.*;
import java.io.*;

class DataCompressor
{
    /* This class provide (more exactly : will provide, as soon as I get the time,) a toolkit to compress data from the WTTransformer and build contains of a compressed file
        - filter with threshold adjustable
        - EZW coding
        - Hufman coding
        - quantification tool
        - header construction

    My purpoes is to compress an image in 5 step:
        1) DWT two-dimesional
        2) Quantification
        3) EZW coding
```

```
    4) Huffman coding
    5) Header insertion
*/

private int threshold = 0;            // parameter to apply at filtering
private int compareMode = 0;            // determine how to compare while filtering
                        //    0 -> relative value
                        //    1 -> absolute value
private int filteringMode = 0;            // determine wich kind of filtering to apply
                        //    0 -> for all x < threshold : x = 0
                        //    1 -> M = mean of datas to filter and for all x < M*threshold : x = 0
                        //    2 -> M = max of datas to filter and for all x < M*threshold : x = 0
                        //    3 -> M = (mean - min) and for all x < M*threshold : x = 0
                        //    4 -> M = (max - min) and for all x < M*threshold : x = 0
                        //    5 -> approximatively the threshold % greater datas will be not set to 0 (lightly slower)
private int infoMode = 0;            // determine what to make with filtering informations. These are always stored in the Vector filteringInfo.
                        //    0 -> nothing more
                        //    1 -> print it to screen
                        //    2 -> print it to file "FilteringInfo.dat"
private Vector filteringInfo = new Vector();    // filtering informations storage, initialized after the filtering
                        // format is : [threshold,
                        //        mean,
                        //        min,
                        //        max,
                        //        (min - mean),
                        //        (min - max),
                        //        écart quadratique moyen (how to say that in english?),
                        //        effective comparing value,
                        //        number of coefficients set to 0,
                        //        percentage of coefficients set to 0,
                        //        compare mode]

public void setFilteringParameters(int thInit, int compareModeInit, int filModeInit, int infoModeInit)
{
    // set the filtering parameters to indicated values
    setThreshold(thInit);
    setCompareMode(compareModeInit);
    setFilteringMode(filModeInit);
    setInfoMode(infoModeInit);
}

public void setThreshold(int thresholdInit)
{
    threshold = thresholdInit;
}

public int getThreshold()
{
    return threshold;
}

public void setCompareMode(int compareModeInit)
{
    compareMode = compareModeInit;
}
```

```
public int getCompareMode()
{
    return compareMode;
}

public void setFilteringMode(int filteringModeInit)
{
    filteringMode = filteringModeInit;
}

public int getFilteringMode()
{
    return filteringMode;
}

public void setInfoMode(int infoModeInit)
{
    infoMode = infoModeInit;
}

public int getInfoMode()
{
    return infoMode;
}

public Vector getFilteringInfo()
{
    return filteringInfo;
}

public int[] ArrayDoubleRounding(double[] input)
{
    // return an array with the same value of input BUT rounded as integer

    int[] output = new int[input.length];

    for (int i = 0; i < input.length; i++)
    {
        output[i] = (int) Math.round(input[i]);
    }

    return output;
}

public double[] ArrayIntConvertion(int[] input)
{
    // return an array with the same value of input but in double format

    double[] output = new double[input.length];

    for (int i = 0; i < input.length; i++)
    {
        output[i] = (double) input[i];
    }

    return output;
```

```
}

public int[] filtering(int[] input)
{
    // filter input with respect to filteringMode value

    if ((compareMode != 0) && (compareMode != 1))
    {
        System.out.println("Unsupported compare mode : no filtering performed");
        return input;
    }

    int[] output = new int[input.length];

    int dim = input.length,mean = 0, max = input[0], min = input[0], minMeanGap = 0, minMaxGap = 0, dummy, test, setToZeroNumber = 0;
    double meanQuadGap = 0;
    float setToZeroPercent = 0;

    for (int i = 0; i < dim; i++)
    {
        if (input[i] > max) max = input[i];
        if (input[i] < min) min = input[i];
        mean += input[i];

        if (filteringMode == 5)
        {
            if (compareMode == 0) output[i] = input[i];
            if (compareMode == 1) output[i] = (int) Math.round(Math.sqrt(input[i] * input[i]));
        }
    }
    mean /= dim;
    minMeanGap = mean - min;
    minMaxGap = max - min;

    switch (filteringMode)
    {

        case 0:
        {
            test = threshold;
            break;
        }
        case 1:
        {
            test = threshold*mean;
            break;
        }
        case 2:
        {
            test = threshold*max;
            break;
        }
        case 3:
        {
            test = threshold*minMeanGap;
            break;
```

```
        }
        case 4:
        {
            test = threshold*minMaxGap;
            break;
        }
        case 5:
        {
            sort(output);
            int boundIndex = dim - (int) Math.round((((float) dim) / 100.) * ((float) threshold));
            test = output[boundIndex];
            break;
        }
        default:
        {
            System.out.println("Unsupported filtering mode : no filtering performed");
            return input;
        }

    }

    for (int i = 0; i < dim; i++)
    {
        if (compareMode == 0)
        {
            if (input[i] >= test)
                output[i] = input[i];
            else
            {
                output[i] = 0;
                setToZeroNumber++;
            }
        }
        else
        {
            if ( ((int) Math.round(Math.sqrt(input[i] * input[i]))) >= test)
                output[i] = input[i];
            else
            {
                output[i] = 0;
                setToZeroNumber++;
            }
        }

        dummy = input[i] - mean;
        meanQuadGap += dummy*dummy;
    }

    meanQuadGap /= dim;
    meanQuadGap = Math.sqrt(meanQuadGap);
    setToZeroPercent = 100 * setToZeroNumber / dim;

    filteringInfo.add(new Integer(threshold));
    filteringInfo.add(new Integer(mean));
    filteringInfo.add(new Integer(min));
    filteringInfo.add(new Integer(max));
```

```
filteringInfo.add(new Integer(minMeanGap));
filteringInfo.add(new Integer(minMaxGap));
filteringInfo.add(new Double(meanQuadGap));
filteringInfo.add(new Integer(test));
filteringInfo.add(new Integer(setToZeroNumber));
filteringInfo.add(new Float(setToZeroPercent));
if(compareMode == 0)
    filteringInfo.add("relative");
else
    filteringInfo.add("absolute");

switch(infoMode)
{
    case 0:
        break;
    case 1:
    {
        System.out.println("threshold : " + threshold);
        System.out.println("mean : " + mean);
        System.out.println("min : " + min);
        System.out.println("max : " + max);
        System.out.println("minMeanGap : " + minMeanGap);
        System.out.println("minMaxGap : " + minMaxGap);
        System.out.println("meanQuadGap : " + meanQuadGap);
        System.out.println("test : " + test);
        System.out.println("setToZeroNumber : " + setToZeroNumber);
        System.out.println("setToZeroPercent : " + setToZeroPercent);
        if(compareMode == 0)
            System.out.println("compare mode : relative");
        else
            System.out.println("compare mode : absolute");
        break;
    }
    case 2:
    {
        try
        {
            PrintWriter out = new PrintWriter(new FileWriter("FilteringInfo.dat"));
            out.println("threshold : " + threshold);
            out.println("mean : " + mean);
            out.println("min : " + min);
            out.println("max : " + max);
            out.println("minMeanGap : " + minMeanGap);
            out.println("minMaxGap : " + minMaxGap);
            out.println("meanQuadGap : " + meanQuadGap);
            out.println("test : " + test);
            out.println("setToZeroNumber : " + setToZeroNumber);
            out.println("setToZeroPercent : " + setToZeroPercent);
            if(compareMode == 0)
                out.println("compare mode : relative");
            else
                out.println("compare mode : absolute");
            out.close();
        }
        catch (IOException e)
        {
```

```
                System.out.println("IOException while printing to file : " + e.getMessage());
            }
            break;
        }
        default:
            System.out.println("Unsupported infoMode");
    }

    return output;
}


/**
 * A quick sort demonstration algorithm
 * SortAlgorithm.java
 *
 * @author James Gosling
 * @author Kevin A. Smith
 * @version     @(#)QSortAlgorithm.java    1.3, 29 Feb 1996
*/

/** This is a generic version of C.A.R Hoare's Quick Sort
   * algorithm.  This will handle arrays that are already
    * sorted, and arrays with duplicate keys.<BR>
    *
    * If you think of a one dimensional array as going from
    * the lowest index on the left to the highest index on the right
    * then the parameters to this function are lowest index or
    * left and highest index or right.  The first time you call
    * this function it will be with the parameters 0, a.length - 1.
    *
    * @param a       an integer array (will be sorted in increasing order)
    * @param lo0     left boundary of array partition
    * @param hi0     right boundary of array partition
    */
   private void QuickSort(int a[], int lo0, int hi0)
   {
          int lo = lo0;
          int hi = hi0;
          int mid;

          if ( hi0 > lo0)
          {

             /* Arbitrarily establishing partition element as the midpoint of
              * the array.
              */
             mid = a[ ( lo0 + hi0 ) / 2 ];

            // loop through the array until indices cross
             while( lo <= hi )
             {
                   /* find the first element that is greater than or equal to
                    * the partition element starting from the left Index.
                    */
                while( ( lo < hi0 ) && ( a[lo] < mid ))
```

```
                    ++lo;

                       /* find an element that is smaller than or equal to
                        * the partition element starting from the right Index.
                        */
                       while( ( hi > lo0 ) && ( a[hi] > mid ))
                      --hi;

                       // if the indexes have not crossed, swap
                       if( lo <= hi )
                       {
                              swap(a, lo, hi);
                              ++lo;
                              --hi;
                       }
                }

            /* If the right index has not reached the left side of array
                  * must now sort the left partition.
                  */
                 if( lo0 < hi )
                        QuickSort( a, lo0, hi );

                 /* If the left index has not reached the right side of array
                  * must now sort the right partition.
                  */
                 if( lo < hi0 )
                        QuickSort( a, lo, hi0 );

        }
    }

       private void swap(int a[], int i, int j)
       {
              int T;
              T = a[i];
              a[i] = a[j];
              a[j] = T;
       }

       private void sort(int a[])
       {
              QuickSort(a, 0, a.length - 1);
       }

}
```


E.1.4 La classe ImageCompressor

```
    package WaveletTools;

import java.io.*;
```

```
import java.math.*;

public class ImageCompressor
{
    /* This class provide a function to compress bmp images with DWT
     * It use of classes WTTransformer, DataCompressor and bitmapHandler
     * The idea is to give an image (bitmap format) and a parameter file to retreive a compressed image
     * I add a feature with the possibility to compress by little blocs of 16x16 or 8x8
     *
     * the parameter file MUST contains :
     *
     *    source file name (something.bmp)
     *    destination file name (somethingElse)
     *
     *                only bmp format is supported (and nevertheless I'm so vexed with this scrappy Redmond format)
     *
     *    compresion mode (0 => entire image in one shot, Z => compression by blocs of ZxZ size - Z MUST be power of two)
     *    motherWavelet (DAUB, HAAR, ...)
     *    ncoef (for DAUB: 4, 12, 20; for other ... see DataWavelet for more informations)
     *    DWT mode (1 => program stop if errors occured during DWT, -1 no)
     *
     *                all these parameters are used by WTTransformer
     *
     *    threshold for filtering (with respect to mode of filtering)
     *    comparing mode (0 => relative value, 1 => absolute value)
     *    filtering mode (0, 1, 2, 3, 4, 5)
     *    filtering info mode (0, 1, 2), if you set compressionMode!=0 and this to 1 or 2, info will be printed only for the last bloc!
     *
     *                all these parameters are used by DataCompressor
     *
     *
     * BEWARE that bad parameters values will cause an error (I hope) or a crash (I fear). Please, read the code of concerned classes
     * to have more informations...
     *
     * here is an example of  parameter file
     *
     *    something.bmp
     *    somethingElse
     *    0
     *    HAAR
     *    007
     *    1
     *    20
     *    1
     *    5
     *    1
     *
     * and nothing else please...
     *
     * I hope I will soon be able to make an attractive interface to avoid this boring procedure but for this time...
     * It's all!
     */

     public static void compressImage(String fileName)
     {
```

```
String sourceFile = "";
String destinationFile = "";
int compressionMode = 0;
int horSize = 0;
int vertSize = 0;
String motherWavelet = "";
int ncoef = 0;
int errorDWTMode = 0;
int threshold = 0;
int compareMode = 0;
int filteringMode = 0;
int infoFilteringMode = 0;

try
{
    BufferedReader param = new BufferedReader(new FileReader(fileName));
    sourceFile = param.readLine().trim();
    destinationFile = param.readLine().trim();
    compressionMode = Integer.parseInt(param.readLine().trim());
    motherWavelet = param.readLine().trim();
    ncoef = Integer.parseInt(param.readLine().trim());
    errorDWTMode = Integer.parseInt(param.readLine().trim());
    threshold = Integer.parseInt(param.readLine().trim());
    compareMode = Integer.parseInt(param.readLine().trim());
    filteringMode = Integer.parseInt(param.readLine().trim());
    infoFilteringMode = Integer.parseInt(param.readLine().trim());
    param.close();
}
catch (IOException e)
{
    System.out.println("Error while reading parameters : " + e.getMessage());
}

BitmapHandler bmph = new BitmapHandler();
System.out.println("Extracting data");
bmph.extractBMPData(sourceFile);                                    // Data extraction
System.out.println("Data extracted");
horSize = (bmph.getSize())[0];
vertSize = (bmph.getSize())[1];
                                                  // powers of 2 ?
if (!powerOf2(horSize) || !powerOf2(vertSize) || ((compressionMode != 0) && (!powerOf2(compressionMode))))
{
    System.out.println("Error: image's dimensions or blocs dimensions are not powers of 2!");
    return;
}

DataCompressor dc = new DataCompressor();
int[] inputI = bmph.getRasterData();
double[] input = dc.ArrayIntConvertion(inputI);
double[] resultF = new double[input.length];                                  // final result init

if (compressionMode == 0)
{
    // compression with all image in one shot

    int[] nn = {horSize, vertSize};
```

```
    dc.setFilteringParameters(threshold, compareMode, filteringMode, infoFilteringMode);
    WTTransformer wtt = new WTTransformer(input, nn, 2, 1, motherWavelet, ncoef, errorDWTMode);
    System.out.println("Performing DWT");
    wtt.wtn();                                          // WTT
    System.out.println("DWT performed");
    double[] result1 = wtt.getOut();
    int[] rounded1 = dc.ArrayDoubleRounding(result1);
    System.out.println("Filtering");
    int[] filteredI = dc.filtering(rounded1);                          // Filtering
    double[] filtered = dc.ArrayIntConvertion(filteredI);
    System.out.println("Filtering performed");
    System.out.println("Performing DWT-1");
    wtt.reinitialize(filtered, nn, 2, -1, motherWavelet, ncoef, errorDWTMode);
    wtt.wtn();                                          // WTT-1
    System.out.println("DWT-1 performed");
    resultF = wtt.getOut();
}
else
{
    // compression by blocs of compressionMode*compressionMode pixels

    int bSize = compressionMode;
    int blocNumber = 0;
    int totalBlocs = ((vertSize / bSize) * (horSize / bSize));
    double[] bloc = new double[bSize*bSize];

    int[] nn = {bSize, bSize};
    dc.setFilteringParameters(threshold, compareMode, filteringMode, 0);          // info filtering mode = 0
    WTTransformer wtt = new WTTransformer(bloc, nn, 2, 1, motherWavelet, ncoef, errorDWTMode);
    System.out.println("Performing compression");

    for (int I = 0; I < (vertSize / bSize); I++)                       // bloc line index
    {
        for (int J = 0; J < (horSize / bSize); J++)                    // bloc column indax
        {

            blocNumber = ((I * (vertSize / bSize)) + J + 1);
            System.out.print("Bloc " + blocNumber + " of " + totalBlocs + "\r");

            for (int i = 0; i < bSize; i++)                        // in bloc line index
            {
                for (int j = 0; j < bSize; j++)                    // in bloc column index
                {
                    bloc[(i * bSize) + j] = input[(I * horSize * bSize) + (J * bSize) + (i * horSize) + j];
                }
            }                                   // local bloc ready

            wtt.reinitialize(bloc, nn, 2, 1, motherWavelet, ncoef, errorDWTMode);
            wtt.wtn();                                  // WTT on local bloc
            double[] result1 = wtt.getOut();
            int[] rounded1 = dc.ArrayDoubleRounding(result1);

            if (blocNumber == totalBlocs)
            {                                   // print filtering info for last bloc
                if (infoFilteringMode == 1)
                    System.out.print("                                    \r");// erase the blocs counter
```

```
                    dc.setFilteringParameters(threshold, compareMode, filteringMode, infoFilteringMode);
                }

                int[] filteredI = dc.filtering(rounded1);                // Filtering local DWT bloc
                double[] filtered = dc.ArrayIntConvertion(filteredI);
                wtt.reinitialize(filtered, nn, 2, -1, motherWavelet, ncoef, errorDWTMode);
                wtt.wtn();                                  // WTT-1
                double[] resultL = wtt.getOut();

                for (int i = 0; i < bSize; i++)                          // filling resultF
                {
                    for (int j = 0; j < bSize; j++)
                    {
                        resultF[(I * horSize * bSize) + (J * bSize) + (i * horSize) + j] = resultL[(i * bSize) + j];
                    }
                }

            }
        }

        System.out.println("Compression performed");
    }

    int[] rounded2 = dc.ArrayDoubleRounding(resultF);

    double meanError = 0;
    int maxError = 0;
    int error = 0;
    for (int i = 0; i < input.length; i++)                                  // errors computation
    {
        error = (int) Math.sqrt((rounded2[i] - inputI[i]) * (rounded2[i] - inputI[i]));
        if (error > maxError)
            maxError = error;
        meanError = meanError + error;
    }
    meanError = meanError / ((double) input.length);
    System.out.println("Maximal error : " + maxError);
    System.out.println("Mean error : " + meanError);

    bmph.setRasterData(rounded2);
    System.out.println("Composing bmp");
    bmph.composeBMP(destinationFile);                                       // BMP construction
    System.out.println("Bmp composed : " + destinationFile + ".bmp");

}

private static boolean power0f2(int n)
{
    if (n == 2)
        return true;
    else if ((n % 2) != 0 )
        return false;
    else
        return power0f2(n / 2);
}
```

```
}
```

E.1.5 La classe DataWavelet

```
     package WaveletTools;

import java.math.*;

/*
    DataWavelet is only a class which contains the coefficients of the supported wavelets families

        daub == Daubechies wavelets (source : "Numerical Recipies in Fortran77", Cambridge University Press, 1992)
        haar == The Haar wavelets

    each wavelet have an xxxC integer array wich contains indexes of the centers of cc and cr:
        for DAUB the centers are ncoef/2

*/

class DataWavelet
{
    private double sqrt2 = Math.sqrt(2.);
    private double[] daub4={0.4829629131445341, 0.8365163037378079, 0.2241438680420134, -0.1294095225512604};
    private double[] daub12={.111540743350, .494623890398, .751133908021, .315250351709, -.226264693965, -.129766867567, .097501605587, .027522865530,-.031582039318,.00055384220
    private double[] daub20={.026670057901, .188176800078, .527201188932, .688459039454, .281172343661,-.249846424327,-.195946274377, .127369340336, .093057364604,-.071394147166
    private double[] haar={.707106781188, .707106781188};
    //private double[] mey={};
    //private double[] mex={};
    //private double[] bat={};
    //private double[] mor={};
    private double[] lbsp = {0.5/sqrt2, 1.0/sqrt2, 0.5/sqrt2, 0.0, 0.0, 1.0/12.0/sqrt2, -0.5/sqrt2, 5.0/6.0/sqrt2, -0.5/sqrt2, 1.0/12.0/sqrt2};
    private double[] burt = {0.0, -.05, .25, .6, .25, -.05, 0., -.010714285714, -.053571428571, .260714285714, .607142857143, .260714285714, -.053571428571, -.010714285714};
    private int[] daub4C={-2, -2};
    private int[] daub12C={-6, -6};
    private int[] daub20C={-10, -10};
    private int[] haarC={-1, -1};
    //private int[] meyC={};
    //private int[] mexC={};
    //private int[] batC={};
    //private int[] morC={};
    private int[] lbspC = {0, 0};
    private int[] burtC = {-3, -3};

    public double[] getData(String name, int ncoef)
    {

        // return the requested array of coefficients or null if unsupported (with a warning message in this case)

        if (name.equals("DAUB"))
        {
            switch (ncoef)
            {
```

```
            case 4:
                return daub4;
            case 12:
                return daub12;
            case 20:
                return daub20;
            default:
                System.out.println("unimplemented value of coefficients number");
        }
    }
    else if (name.equals("HAAR"))return haar;
    //else if (name.equals("MEY")) return mey;
    //else if (name.equals("MEX")) return mex;
    //else if (name.equals("BAT")) return bat;
    //else if (name.equals("MOR")) return mor;
    else if (name.equals("LBSP")) return lbsp;
    else if (name.equals("BURT")) return burt;
    else
    {
        System.out.println("unimplemented set of wavelets");
    }

    return null;
}

public int[] getCenter(String name, int ncoef)
{

    // return the requested conventionnal center of cc and cr or null if unsupported (with a warning message in this case)

    if (name.equals("DAUB"))
    {
        switch (ncoef)
        {
            case 4:
                return daub4C;
            case 12:
                return daub12C;
            case 20:
                return daub20C;
            default:
                System.out.println("unimplemented value of coefficients number");
        }
    }
    else if (name.equals("HAAR")) return haarC;
    //else if (name.equals("MEY")) return meyC;
    //else if (name.equals("MEX")) return mexC;
    //else if (name.equals("BAT")) return batC;
    //else if (name.equals("MOR")) return morC;
    else if (name.equals("LBSP")) return lbspC;
    else if (name.equals("BURT")) return burtC;
    else
    {
        System.out.println("unimplemented set of wavelets");
    }
```

```
        return null;
    }
}
```

E.1.6 La classe WaveHandler

```
    package WaveletTools;

import java.io.*;
import java.util.Vector;

class WaveHandler
{
    /* This class provide a toolkit to handle wave files.
     * BEWARE I choose to store Sizes in integer (4 bytes) -> the sizes are limited to 2^31 !!!!!!!
     * at 22kHz, record can play 27 hours
       */

    // wave format parameters

    // 1) header (12 bytes)

    private int[] riff = {82, 73, 70, 70};           // 'RIFF' in iso latin ascii chart
    private int[] fileSize = new int[4];             // 4 bytes
    private int[] wave = {87, 65, 86, 69};           // 'WAVE' in iso latin ascii chart

    // 2) info header (32 bytes)

    private int[] fmt = {102, 109, 116, 32};        // 'fmt ' in iso latin ascii chart
    private int[] formatChunklenght = {16, 0, 0, 0};    // ?
    private int[] monoStereo = new int[2];           // mono => 0; stereo => 1
    private int[] channelNumber = new int[2];        // ?
    private int[] sampleRate = new int[4];           // sample rate (in Hz)
    private int[] bytesPerSecond = new int[4];       // it isn't clear enough?
    private int[] bytesPerSample = new int [2];       // 1 = 8 bit mono; 2 = 8 bit stereo or 16 bit mono; 4 = 16 bit stereo
    private int[] bitPerSample = new int[2];        // why? Contains any new information?

    private int[] data = {100, 97, 116, 97};        // 'data' in iso latin ascii chart
    private int[] dataSize = new int[4];            // size of the data

    private int totHeadSize = 44;                // total length of headers : 44 bytes

    private Vector param = null;                // this vector must contains all parameter above \huge {IN GOOD ORDER!!!}
                                // it is fill in standart constructor -> this constructor must be update in case of format changes
    // parameters functions

    public WaveHandler()
    {
        // constructor only fill param with all parameter for re-use later
        // beware to update this method if some format changes are made...

        param = new Vector();
```

```
        param.add(riff);
        param.add(fileSize);
        param.add(wave);
        param.add(fmt);
        param.add(formatChunklenght);
        param.add(monoStereo);
        param.add(channelNumber);
        param.add(sampleRate);
        param.add(bytesPerSecond);
        param.add(bytesPerSample);
        param.add(bitPerSample);
        param.add(data);
        param.add(dataSize);
    }

    public void setFileParameters(String file)
    {
        // set all the current parameters with those of file (it should be a .wave)

        try
        {
            FileInputStream wave = new FileInputStream(file);

            int[] parameter = null;
            for (int i = 0; i < param.size(); i++)
            {
                parameter = (int[]) param.get(i);
                for (int j = 0; j < parameter.length; j++)
                {
                    parameter[j] = wave.read();
                }
            }
        }
        catch (IOException e)
        {
            System.out.println("IOException occured : " + e.getMessage());
        }

    }

    // working attributes

    String fileName = "";
    int[] rasterData1 = null;           // channel mono or channel 1 stereo
    int[] rasterData2 = null;           // channel 2 stereo (unused in mono)

    // handling functions

    public int getMonoStereo()
    {
        return monoStereo[0];
    }

    public int[] getRasterData1()
    {
```

```
    return rasterData1;
}

public int[] getRasterData2()
{
    return rasterData2;
}

public void setRasterData1(int[] data)
{
    // set rasterData1 values to those of data

    rasterData1 = new int[data.length];
    for (int i= 0; i < data.length; i++)
    {
        rasterData1[i] = data[i];
    }
}

public void setRasterData2(int[] data)
{
    // set rasterData2 values to those of data

    rasterData2 = new int[data.length];
    for (int i= 0; i < data.length; i++)
    {
        rasterData2[i] = data[i];
    }
}

public int getSize()
{
    // return the number of SAMPLES in record!!!!

    return rasterData1.length;
}

public void extractWaveData(String file)
{
    // extract data fom wave file to rasterData1 and rasterData2 (only for stereo records)
    // put file Name to fileName
    // set parameters to those of the file

    fileName = file;
    setFileParameters(file);

    int length = dataSize[0] + (dataSize[1] * 256) + (dataSize[2] * 65536) + (dataSize[3] * 16777216);

    if (bytesPerSample[0] == 1)
        rasterData1 = new int[length];            // 8 bits mono
    else if ((bytesPerSample[0] == 2) && (monoStereo[0] == 0))
        rasterData1 = new int[(length / 2)];        // 16 bits mono
    else if (bytesPerSample[0] == 2)
    {
        rasterData1 = new int[(length / 2)];        // 8 bits stereo
        rasterData2 = new int[(length / 2)];
```

```
}
else
{
    rasterData1 = new int[(length / 4)];        // 16 bits stereo
    rasterData2 = new int[(length / 4)];
}

int data1 = 0;
int data2 = 0;

try
{
    FileInputStream wave = new FileInputStream(file);

    // goto Raster data

    for (int i = 0; i < totHeadSize; i++)
    {
        wave.read();
    }

    // read Raster data

    for (int i = 0; i < rasterData1.length; i++)
    {

        data1 = wave.read();

        if (bytesPerSample[0] == 1)
            rasterData1[i] = data1;                    // 8 bits mono
        else if ((bytesPerSample[0] == 2) && (monoStereo[0] == 0))
        {
            data2 = wave.read();
            rasterData1[i] = data1 + (data2 * 256);           // 16 bits mono
        }
        else if (bytesPerSample[0] == 2)
        {
            data2 = wave.read();
            rasterData1[i] = data1;                    // 8 bits stereo
            rasterData2[i] = data2;
        }
        else
        {
            data2 = wave.read();                    // 16 bits stereo
            rasterData1[i] = data1 + (data2 * 256);
            data1 = wave.read();
            data2 = wave.read();
            rasterData2[i] = data1 + (data2 * 256);
        }

    }

    wave.close();
}
catch (IOException e)
{
```

```
            System.out.println("IOException occured : " + e.getMessage());
        }

    }

    public void composeWave(String file)
    {
        // compose a file named file.wav in wave format with rasterData1 (and rasterData2 if stereo) and current parameters
        // put file.wav to fileName

        fileName = file + ".wave";

        try
        {
            FileOutputStream wave = new FileOutputStream(file + ".wav");

            // print header

            int[] parameter = null;
            for (int i = 0; i < param.size(); i++)
            {
                parameter = (int[]) param.get(i);
                for (int j = 0; j < parameter.length; j++)
                {
                    wave.write(parameter[j]);
                }
            }

            // print Raster data

            for (int i = 0; i < rasterData1.length; i++)
            {

                if (bytesPerSample[0] == 1)
                    wave.write(rasterData1[i]);                // 8 bits mono
                else if ((bytesPerSample[0] == 2) && (monoStereo[0] == 0))
                {
                    wave.write(rasterData1[i] % 256);            // 16 bits mono
                    wave.write(rasterData1[i] / 256);
                }
                else if (bytesPerSample[0] == 2)
                {
                    wave.write(rasterData1[i]);                // 8 bits stereo
                    wave.write(rasterData2[i]);
                }
                else
                {
                    wave.write(rasterData1[i] % 256);            // 16 bits stereo
                    wave.write(rasterData1[i] / 256);
                    wave.write(rasterData2[i] % 256);
                    wave.write(rasterData2[i] / 256);
                }

            }

            wave.close();
```

```
        }
        catch (IOException e)
        {
            System.out.println("IOException occured : " + e.getMessage());
        }

    }
}
```

E.1.7 La classe WaveCompressor

```
    package WaveletTools;

import java.io.*;
import java.math.*;

public class WaveCompressor
{
    /* This class provide a function to compress wave file with DWT
     * It use of classes WTTransformer, DataCompressor and WaveHandler
     * The idea is to give an record (RIFF format) and a parameter file to retreive a compressed record
     *
     * the parameter file MUST contains :
     *
     *    source file name (something.wav)
     *    destination file name (somethingElse)
     *
     *                only wave format is supported
     *
     *    motherWavelet (DAUB, HAAR, ...)
     *    ncoef (for DAUB: 4, 12, 20; for other ... see DataWavelet for more informations)
     *    DWT mode (1 => program stop if errors occured during DWT, -1 no)
     *
     *                all these parameters are used by WTTransformer
     *
     *    threshold for filtering (with respect to mode of filtering)
     *    comparing mode (0 => relative value, 1 => absolute value)
     *    filtering mode (0, 1, 2, 3, 4, 5)
     *    filtering info mode (0, 1, 2), if you set compressionMode!=0 and this to 1 or 2, info will be printed only for the last bloc!
     *
     *                all these parameters are used by DataCompressor
     *
     *
     * BEWARE that bad parameters values will cause an error (I hope) or a crash (I fear). Please, read the code of concerned classes
     * to have more informations...
     *
     * here is an example of  parameter file
     *
     *    something.bmp
     *    somethingElse
     *    HAAR
     *    007
```

```
*    1
*    20
*    1
*    5
*    1
*
* and nothing else please...
*
* I hope I will soon be able to make an attractive interface to avoid this boring procedure but for this time...
* It's all!
*/

public static void compressWave(String fileName)
{

   String sourceFile = "";
   String destinationFile = "";
   int horSize = 0;
   int vertSize = 0;
   String motherWavelet = "";
   int ncoef = 0;
   int errorDWTMode = 0;
   int threshold = 0;
   int compareMode = 0;
   int filteringMode = 0;
   int infoFilteringMode = 0;

   try
   {
       BufferedReader param = new BufferedReader(new FileReader(fileName));
       sourceFile = param.readLine().trim();
       destinationFile = param.readLine().trim();
       motherWavelet = param.readLine().trim();
       ncoef = Integer.parseInt(param.readLine().trim());
       errorDWTMode = Integer.parseInt(param.readLine().trim());
       threshold = Integer.parseInt(param.readLine().trim());
       compareMode = Integer.parseInt(param.readLine().trim());
       filteringMode = Integer.parseInt(param.readLine().trim());
       infoFilteringMode = Integer.parseInt(param.readLine().trim());
       param.close();
   }
   catch (IOException e)
   {
       System.out.println("Error while reading parameters : " + e.getMessage());
   }

   WaveHandler wh = new WaveHandler();
   System.out.println("Extracting data");
   wh.extractWaveData(sourceFile);                                    // Data extraction
   System.out.println("Data extracted");

   DataCompressor dc = new DataCompressor();

   int[][] data = {null, null};
   data[0] = wh.getRasterData1();
   if (wh.getMonoStereo() == 1)
```

```
    data[1] = wh.getRasterData2();


int[] result = new int[data[0].length];
int i = 0;
while ((i < 2) && (data[i] != null))
{

    int dim = (int) Math.pow(2, 30);               // int goes from -2^31 to (2^31 - 1)
    int r = data[i].length;
    int x = 0;
    int index = 0;

    while (dim >= 1)                        // slicing of the datas into segment of lengths of
    {                                  // powers of 2 : dim
        x = r / dim;
        r = r % dim;

        if (x != 0)
        {

            System.out.println("Bloc of length : " + dim);

            if (dim == 1)
            {
                result[(data[i].length - 1)] = data[i][(data[i].length - 1)];
            }
            else
            {
                int[] dataLocal = new int[dim];

                for (int j = 0; j < dim; j++)                                // copy bloc
                {
                    dataLocal[j] = data[i][index + j];
                }

                double[] inputL = dc.ArrayIntConvertion(dataLocal);
                double[] resultL = new double[inputL.length];                        // final result init
                int[] nn = {inputL.length};
                dc.setFilteringParameters(threshold, compareMode, filteringMode, infoFilteringMode);
                WTTransformer wtt = new WTTransformer(inputL, nn, 1, 1, motherWavelet, ncoef, errorDWTMode);
                wtt.wtn();                                          // WTT
                double[] result1 = wtt.getOut();
                int[] rounded1 = dc.ArrayDoubleRounding(result1);
                int[] filteredI = dc.filtering(rounded1);                          // Filtering
                double[] filtered = dc.ArrayIntConvertion(filteredI);
                wtt.reinitialize(filtered, nn, 1, -1, motherWavelet, ncoef, errorDWTMode);
                wtt.wtn();                                          // WTT-1
                resultL = wtt.getOut();
                int[] rounded2 = dc.ArrayDoubleRounding(resultL);

                for (int j = 0; j < dim; j++)                                // copy back
                {
                    result[index + j] = rounded2[j];
                }
            }
```

```
                index = index + dim;

            }
            dim = dim / 2;
        }

        double meanError = 0;
        int maxError = 0;
        int error = 0;
        for (int j = 0; j < data[i].length; j++)                                    // errors computation
        {
            error = (int) Math.sqrt((result[j] - data[i][j]) * (result[j] - data[i][j]));
            if (error > maxError)
                maxError = error;
            meanError = meanError + error;
        }
        meanError = meanError / ((double) data[i].length);
        System.out.println("Maximal error : " + maxError);
        System.out.println("Mean error : " + meanError);

        if (i == 0)
            wh.setRasterData1(result);
        else if (i == 1)
            wh.setRasterData2(result);
        else
            System.out.println("Error channel !!!???!!!");

        i++;
    }

    System.out.println("Composing wave");
    wh.composeWave(destinationFile);                                    // Wave construction
    System.out.println("Wave composed : " + destinationFile + ".wav");

  }

}
```


E.2 Le programme FORTRAN90

```
 module filtering

implicit none
real, private :: minD, maxD, meanD, threshold, minMaxGap, meanQuadGap, setToZeroPercent
integer, private :: nmb, setToZeroNmb
real, allocatable, dimension(:), private :: data2Filter

CONTAINS

subroutine setFilteringParameters(dat ,th, type)
```

```
    implicit none

    real, dimension(:), intent(in) :: dat
    real, intent(in) :: th
    integer, intent(in) :: type

    nmb = size(dat)

    allocate(data2Filter(nmb))

    data2Filter = dat

    minD = minval(data2Filter)
    maxD = maxval(data2Filter)
    meanD = sum(data2Filter) / nmb
    minMaxGap = maxD - minD

    select case (type)

        case (1)
            threshold = th
        case (2)
            threshold = th * minD
        case (3)
            threshold = th * maxD
        case (4)
            threshold = th * meanD
        case (5)
            threshold = th * minMaxGap
        case default
            print *,'Error: Unsupported filtering mode : ', type
            stop

    end select

    meanQuadGap = 0.
    setToZeroPercent = 0
    setToZeroNmb = 0

    return

end subroutine setFilteringParameters

subroutine filter()

    ! execute filtering !! BEWARE that all parameters MUST be set !!

    implicit none

    integer :: i
    real :: dummy

    do i = 1, nmb

        dummy = data2Filter(i) - meanD
```

```
            meanQuadGap = dummy * dummy

            if (sqrt(data2Filter(i) * data2Filter(i)) <= threshold) then
                data2Filter(i) = 0.
                setToZeroNmb = setToZeroNmb + 1
            end if
        end do

        meanQuadGap = sqrt(meanQuadGap / real(nmb))
        setToZeroPercent = 100. * real(setToZeroNmb) / real(nmb)

        print *,'min: ', minD
        print *,'max: ', maxD
        print *,'mean: ', meanD
        print *,'threshold: ', threshold
        print *,'mean square gap: ', meanQuadGap
        print *,'number of coef. set to 0: ', setToZeroNmb, ' -> ',setToZeroPercent, '%'

        return

    end subroutine filter

    subroutine getFilteringResult(dat)

        implicit none

        real, dimension(:), intent(inout) :: dat
        integer :: i

        if (size(dat) /= nmb) then
            print *,'Error: uncompatible array'
        else
            do i = 1, nmb
                dat(i) = data2Filter(i)
            end do
        end if

        return

    end subroutine getFilteringResult

end module filtering

module WTData

    implicit none

    real, dimension(4), private, parameter :: c4 = (/0.4829629131445341, 0.8365163037378079,0.2241438680420134, &
                          -0.1294095225512604/)
    real, dimension(12), private, parameter :: c12 = (/.111540743350, .494623890398, .751133908021, &
                         .315250351709,-.226264693965,-.129766867567,.097501605587, &
                         .027522865530,-.031582039318,.000553842201, .004777257511, &
                            -.001077301085/)
        real, dimension(20), private, parameter :: c20 = (/.026670057901, .188176800078, .527201188932, &
                         .688459039454, .281172343661,-.249846424327,-.195946274377, &
                              .127369340336, .093057364604,-.071394147166,-.029457536822, &
```

```
                              .033212674059,.003606553567,-.010733175483, .001395351747, &
                              .001992405295,-.000685856695,-.000116466855,.000093588670, &
                              -.000013264203 /)
    real, dimension(2), private, parameter ::  h2 = (/.707106781188, .707106781188/)
    real, dimension(9), private, parameter :: s1 = (/    0.0331456289, -0.0662912577, -0.176776692,  0.419844657,  0.994368911,&
                                  0.419844657, -0.176776692, -0.0662912577,  0.0331456289 /)
    real, dimension(3), private, parameter :: s1tilde = (/ 0.353553385,  0.707106769,  0.353553385/)
    real, Dimension(12), private, parameter :: s2 = (/ 0.542, 0.307, -0.035, -0.078, 0.023, -0.030, 0.012, &
                               -0.013, 0.006, 0.006, -0.003, -0.002/)


    public :: getData

CONTAINS

subroutine getData(name, nmbr, dat)

    ! put the requested coef. in dat array.
    ! dat MUST be of dimension = nmbr !
    ! name ::= DAUB / HAAR (at this time)
    ! nmbr ::= 2 (for HAAR)/ 4, 12 or 20 (for DAUB)

    character (len = 4), intent(in) :: name
    integer, intent(in) :: nmbr
    real, dimension(nmbr), intent(inout) :: dat
    integer :: i

    if (name == 'DAUB') then

        choix: select case(nmbr)

            case(4) choix

                do i = 1, nmbr
                    dat(i) = c4(i)
                end do

            case(12) choix

                do i = 1, nmbr
                    dat(i) = c12(i)
                end do

            case(20) choix

                do i = 1, nmbr
                    dat(i) = c20(i)
                end do

            case default choix

                print *,'Error : unsupported coef. number for DAUB wavelet'
                stop

        end select choix
```

```
else if (name == 'HAAR') then

    if (nmbr == 2) then

        do i = 1, nmbr
            dat(i) = h2(i)
        end do

    else

        print *,'Error : HAAR wavelet have only 2 coef.'
        stop

    endif

else if (name == 'SPL1') then

    if (nmbr == 9) then

        do i = 1, nmbr
            dat(i) = s1(i)
        end do

    else

        print *,'Error : Spline1 wavelet have only 9 coef.'
        stop

    endif

else if (name == 'SP1T') then

    if (nmbr == 3) then

        do i = 1, nmbr
            dat(i) = s1tilde(i)
        end do

    else

        print *,'Error : Spline1-tilde wavelet have only 3 coef.'
        stop

    endif

else if (name == 'SPL2') then

    if (nmbr == 12) then

        do i = 1, nmbr
            dat(i) = s2(i)*sqrt(2.)
        end do

    else

        print *,'Error : Spline2 wavelet have only 12 coef.'
```

```
            stop

        endif

    else

        print *,'Error : unrecognize mother wavelet'
        stop

    end if

end subroutine getData

end module WTData


module DWTransformer

    use WTData
    implicit none

!    DWTransformer provide a tool to apply wavelet transform in n dimensions with one of the supported wavelets family. I try to make it
!    fully configurable and easy to use.

!    DWTransformer is partialy a made of functions from "Numerical Recipies in Fortran77" (Cambridge University Press 1992).

!    parameter comments:
!        ndim :          the tranformer perform an ndim-dimensional WT.
!        nn :          the size of each dimension of input data (i.e. the real number of values in each dimension); dimension of nn is ndim
!                and each component MUST all be powers of 2!
!        in :          input array; This array have a $ size = \prod_{i=1}^ndim nn[i] $ (\LaTeX format ;-) and the data are store in
!                respect to invert order of nn (as arrays are stored in FORTRAN). That means stored by columns for a two dimensionnal
!                array.
!        out :          output data, stored as input array in.
!        cc :          array that contains the coefficients for the wavelet family choosed by user (initialized by pwtSet). cc is
!                the smoothing filter.
!        cr :          cr is the quadrature mirror filter of cc needed by WT algorithm.
!        ioff and joff:      index of the centering values of cc and cr
!        isign :      if isign=1 wtn perform direct wavelet transform, if isign=-1 wtn perform inverse wavelet transform
!        motherWT :      the conventionnal name of the mother wavelet user wants to use.
!                The supported mother wavelets are (at this time):
!                    Daubechies :      DAUB (4, 12, 20)
!                    The Haar :         HAAR
!        ncoef :      the number of coefficients the user wants to use (ncoef = 4, 12 or 20 for Daubechies wavelets).


integer :: ndim, joff, ioff, isign, ncoef
integer (kind = 4), allocatable, dimension(:) :: nn
real, allocatable, dimension(:) :: in, out, cc, cr
character (len = 4) :: motherWT
real, allocatable, dimension(:) :: wksp_b
real, allocatable, dimension(:) :: wksp

private :: powerOf2
```

```
CONTAINS

subroutine  DWT(dat, n, name, ncof, is)

! Only procedure you should use, DWT perform the entire process of DWT

    implicit none

    character (len = 4), intent(in) :: name
    integer, intent(in) :: ncof
    real, intent(inout), dimension(:) :: dat
    integer, intent(in), dimension(:) :: n
    integer, intent(in) :: is

    integer :: i

    call initialize(dat, n, name, ncof, is)
    call wtn()

    do i = 1, size(dat)
        dat(i) = in(i)
    end do

end subroutine DWT


subroutine initialize(dat, n, name, ncof, is)

    implicit none

! initialize all the working parameters values with respect of input values

    character (len = 4), intent(in) :: name
    integer, intent(in) :: ncof
    real, intent(in), dimension(:) :: dat
    integer, intent(in), dimension(:) :: n
    integer, intent(in) :: is

    call setCoef(name, ncof)
    call setData(dat, n)
    call setIsign(is)

    return

end subroutine initialize


subroutine setCoef(name, ncof)

! set the parameters  cc, cr, ncoef, ioff, joff, motherWT values to requested values

    implicit none

    character (len = 4), intent(in) :: name
    integer, intent(in) :: ncof
    INTEGER :: k
```

```
    REAL :: sig

    if (ncof > 0) then

        if (allocated(cc)) deallocate(cc)
        if (allocated(cr)) deallocate(cr)
        ncoef = ncof
        allocate(cc(ncoef))
        allocate(cr(ncoef))
        call getData(name, ncof, cc)
        motherWT = name

    else

        print *,'Error : number of coef. must be > 0'
        stop

    end if

    sig=-1.

    do k=1,ncoef

            cr(ncoef+1-k) = sig * cc(k)
            sig = -sig

    end do

    ioff = -ncoef / 2
    joff = -ncoef / 2

    return

end subroutine setCoef

subroutine setData(dat, n)

    implicit none

! set the data to dat's values after some checks
! reset the out and working arrays

    real, intent(in), dimension(:) :: dat
    integer, intent(in), dimension(:) :: n
    integer :: sizeDat, sizen, i, k

    sizeDat = size(dat)
    sizen = size(n)
    k = 1

    if(allocated(in)) deallocate(in)
    if(allocated(out)) deallocate(out)
    if(allocated(nn)) deallocate(nn)

    do i = 1, sizen
```

```
        if (powerOf2(n(i))) then

            k = k * n(i)

        else

            print *,'Error : all input dimensions MUST be powers of 2'
            stop

        end if

    end do

    if ( k == sizeDat ) then

        allocate(in(sizeDat))
        allocate(out(sizeDat))
        allocate(nn(sizen))
        in = dat
        out = 0.
        ndim = sizen
        nn = n

    else

        print *,'Error : incompatible input arrays'
        stop

    end if

    return

end subroutine setData

subroutine setIsign(is)

    implicit none

    integer, intent(in) :: is

    if ((is*is) == 1) then

        Isign = is

    else

        print *,'Error: Isign must be 1 or -1'
        stop

    end if

end subroutine setIsign


function powerOf2(nmbr)
```

```
    ! check if an integer is equal to a power of 2

    integer, intent(in) :: nmbr
    logical :: powerOf2
    integer :: dummy

    dummy = nmbr

    powerOf2 = .false.

    do while (dummy > 1)

        powerOf2 = .false.

        if (mod(dummy, 2) == 0) then
            powerOf2 = .true.
            dummy = dummy / 2
        else
            return
        end if

    end do

    return

end function powerOf2

SUBROUTINE wtn()

    INTEGER :: i1,i2,i3,idim,k,n,nnew,nprev,nt,ntot

    allocate(wksp(size(in)))
    allocate(wksp_b(size(in)))

    ntot = 1
    do idim = 1, ndim
        ntot = ntot * nn(idim)
    end do

    nprev = 1

    do idim = 1, ndim

        n = nn(idim)
        nnew = n * nprev

        if (n > 4) then

            do i2 = 0, ntot-1, nnew

                do i1 = 1, nprev

                    i3 = i1 + i2

                    do k = 1, n
                        wksp(k) = in(i3)
```

```
                    i3 = i3 + nprev
                end do

                if (isign >= 0) then

                    nt = n

                    do while (nt >= 4)
                        call pwt(wksp,nt,isign)
                        nt = nt / 2
                    end do

                else

                    nt = 4

                    do while (nt <= n)
                        call pwt(wksp,nt,isign)
                        nt = nt * 2
                    end do

                endif

                i3 = i1 + i2

                do k = 1, n
                    in(i3) = wksp(k)
                    i3 = i3 + nprev
                end do

            end do

        end do

    end if

        nprev=nnew

end do

deallocate(wksp)
deallocate(wksp_b)

return

END subroutine wtn


SUBROUTINE pwt(a,n,isign)

    implicit none

    real, intent(inout), dimension(:) :: a
    integer, intent(in) :: n, isign

    INTEGER i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod
```

```
REAL ai,ai1

if (n < 4) return

nmod = ncoef * n
n1 = n - 1
nh = n / 2

if (isign >= 0) then

    ii = 1

        do i = 1, n, 2

        ni = i + nmod + ioff
        nj = i + nmod + joff

        do k = 1, ncoef
            jf = iand(n1, ni + k)
            jr = iand(n1, nj + k)
            wksp_b(ii) = wksp_b(ii) + cc(k) * a(jf + 1)
            wksp_b(ii + nh) = wksp_b(ii + nh) + cr(k) * a(jr + 1)
        end do

        ii = ii + 1

    end do

else

    ii = 1

    do i = 1, n, 2

        ai = a(ii)
        ai1 = a(ii + nh)
        ni = i + nmod + ioff
        nj = i + nmod + joff

        do k = 1, ncoef
            jf = iand(n1, ni + k) + 1
            jr = iand(n1, nj + k) + 1
            wksp_b(jf) = wksp_b(jf) + cc(k) * ai
            wksp_b(jr) = wksp_b(jr) + cr(k) * ai1
        end do

        ii = ii + 1

    end do

endif

do j = 1, n
    a(j) = wksp_b(j)
    wksp_b(j) = 0.
end do
```

```
    return

END subroutine pwt

subroutine quantif256(a, b)

    real, dimension(:) :: a
    integer (kind = 2), dimension(:) :: b
    integer i

    do i = 1, size(a)
        b(i) = anint(a(i))
        if (b(i) > 255) then
            !print *,'b(',i,') = ',b(i),'set to 255'
            b(i) = 255
        else if (b(i) < 0) then
            !print *,'b(',i,') = ',b(i),'set to 0'
            b(i) = 0
        end if
    end do

    return

end subroutine quantif256

end module DWTransformer


Module BitmapHandler

implicit none

! This Module provide a toolkit to handle bmp image (windows format).
! At this time only one encoding scheme is supported : 8 bits per pixel
! and no compression (I hope I'll soon be able to program other
! encoding scheme -6 in total-)

!    FIXME : SUPPORT ALL FORMAT AT LEAST IN THE READING ROUTINE

! Bitmap windows file format:
! signature 2 bytes = 66 77 (BM in ascii chart)
! 4 bytes reserved (= 0)
! dataOffset 4 bytes (offset to data)
! infoHeader 40 bytes
!    size 4 bytes (size of infoHeader = 40)
!    width 4 bytes (bitmap width)
!    height 4 bytes (bitmap height)
!    planes 2 bytes (number of planes = 1)
!    bitCount 2 bytes (bits per pixel)
!    compression 4 bytes (type of compression)
!    imageSize 4 bytes (compressed size of image -- 0 means that compression = 0)
!    Xresolution 4 bytes (horizontal resolution -- pixels/meter)
!    Yresolution 4 bytes (vertical resolution -- pixels/meter)
!    colorUsed 4 bytes
!    colorImportant 4 bytes (0 = all)
```

```
! colorTable 4 * colorUsed bytes present only if Info.bitCount .le. 8
! colors should be ordered by importance, each color take 4 bytes:
!    Red 1 bytes
!    Green 1 bytes
!    Blue 1 bytes
!    reserved 1 bytes (= 0, will be alpha in the future)
!
! Pixels are store bottom-up, left-to-right. Pixels lines are padded with zeros to end on a 32 bit boundary.
! Color indices are zero based, meaning a pixel color of 0 represent the first color table entry, a pixel color
! of 255 represent the 255th entry.
! There is no color table for images with more then 256 colors.

! The only compression scheme supported is bitCount = 8, compression = 0 with grey levels colorTable
!(each of the 256 colors is [x x x 0] with x = 0, 1, ... , 256)


integer (kind = 4), private, dimension (2), parameter :: signature = (/66, 77/)
integer (kind = 4), private, parameter :: reserved = 0
integer (kind = 4), private, parameter :: infoHeaderSize = 40
integer (kind = 4), private, parameter :: planes = 1
integer (kind = 4), private, parameter :: compressionSupported = 0
integer (kind = 4), private, parameter :: bitCountSupported = 8
integer (kind = 4), private, parameter :: widthIndex = 19
integer (kind = 4), private, parameter :: heightIndex = 23
integer (kind = 4), private, parameter :: ISIndex = 35
integer (kind = 4), private, parameter :: bitCountIndex = 29
integer (kind = 4), private, parameter :: compressionIndex = 31
integer, private, parameter :: ths = 54     ! Total Header Size = 54 bytes
integer, private :: nc = 256              ! Number of Colors
integer, private :: cts = 256 * 4          ! Total size of the Color Table
integer, private, parameter :: ncomp = 4    ! number of color's components

integer (kind = 4), private :: width = 0
integer (kind = 4), private :: height = 0
integer (kind = 4), private :: line = 0
integer (kind = 4), private :: bitcount = 0
integer (kind = 4), private :: compression = 0
integer (kind = 4), private :: imageSize = 0
integer (kind = 4), private :: RealImageSize = 0

integer (kind = 1), private, dimension (ths) :: header
integer (kind = 1), private, allocatable, dimension (:, :) :: colorTable    ! colors are stored in columns
integer (kind = 1), private, allocatable, dimension (:) :: rasterData
integer (kind = 2), private, allocatable, dimension (:) :: image
character (len = 20), private :: fileName
integer, private :: nerr = 0
integer, private :: ierr = 0

public :: extractHeaderInfo, extractColorTable, extractRasterData, composeBMP, getImageData, setImageData, getImageSize
private :: f2o, checkColorTable, checkBMPType, unsigned, signed, errorFile, errorAlloc

CONTAINS

function getImageSize()

    implicit none
```

```
    integer, dimension(3) :: getImageSize
    getImageSize(1) = width
    getImageSize(2) = height
    getImageSize(3) = size(image)

    return

end function getImageSize


subroutine getImageData(dat)

    implicit none
    integer (kind = 2), dimension (size(image)) :: dat

    dat = image

    return

end subroutine getImageData

subroutine setImageData(in)
    implicit none
    integer (kind = 2), dimension (size(image)) :: in

    image = in

end subroutine setImageData

subroutine extractHeaderInfo(file)

    ! read the header and store it into header array
    ! put fileName at file
    ! store informations into corresponding variables (compression, bitcount, Imagesize (with and without padding), width, height, line, nc, cts)

    implicit none
    character (len=20), intent (in) :: file

    fileName = file

    open(10, file = file, status = 'old', err = 100, iostat = nerr, form = 'unformatted', access = 'direct', recl = ths)
    read(10, rec = 1, err = 100, iostat = nerr) header(:)
    close(10)

!    print '(I5)',header(:)

    bitCount = header(bitCountIndex)
    compression = header(compressionIndex)

    if (compression /= 0) then
        print *, 'Error : No compression supported (compression = ', compression,')'
        stop
    endif

    imageSize = f2o(unsigned(header(ISIndex)), unsigned(header(ISIndex + 1)), &
```

```
            unsigned(header(ISIndex + 2)), unsigned(header(ISIndex + 3)))
        width = f2o(unsigned(header(widthIndex)), unsigned(header(widthIndex + 1)), &
            unsigned(header(widthIndex + 2)),unsigned(header(widthIndex + 3)))
        height = f2o(unsigned(header(heightIndex)), unsigned(header(heightIndex + 1)), &
            unsigned(header(heightIndex + 2)), unsigned(header(heightIndex + 3)))
        RealImageSize = imageSize - (height * (mod(width, 4)))            ! total number of pixels without padding
        line = width - (mod(width, 4))                          ! number of pixels per line without padding


        print *,'bitCount: ', bitCount, '; compression: ', compression, '; image size: ', imageSize, '; width', width, &
        '; height: ', height, '; real image size: ', RealImageSize, '; line: ', line

        return

100     call errorFile()

end subroutine extractHeaderInfo

subroutine extractColorTable(file)

    ! read the color table if exist (return an error if there is no such table => bitCount > 8)

    implicit none
    character (len = 20) :: file

    call extractHeaderInfo(file)

    nc = 2 ** bitCount
    if (bitcount <= 8) then
        cts = nComp * nc
        allocate(colorTable(nComp, nc))        ! [stat=] statement does not seem to be supported by the VAST-F90 compiler

        open(10, file = fileName, status = 'old', err = 100, iostat = nerr, &
            form = 'unformatted', access = 'direct', recl = ths + cts)
        read(10, rec = 1, err = 100, iostat = nerr) header(:), colorTable(:, :)    ! BEWARE THAT COLORS ARE STORED IN COLUMNS !!!
        close(10)

!       print *,colorTable(:, :)
    else
        cts = 0
        print *,'Warning: No color table for this file'
    end if

    return

100     call errorFile()
200    call errorAlloc()

end subroutine extractColorTable

subroutine extractRasterData(file)

    ! read the image's pixels and sotre it into rasterData array, store the unsigned values in image array without padding
    ! image contains the data left-to-right, by columns, up-to-bottom

    implicit none
```

```
    character (len = 20) :: file
    integer :: i, j, colBegin

    call extractColorTable(file)

    allocate(rasterData(imageSize))

    open(10, file = fileName, status = 'old', err = 100, iostat = nerr, &
        form = 'unformatted', access = 'direct', recl = ths + cts + imageSize)

    if (allocated(colorTable)) then
        read(10, rec = 1, err = 100, iostat = nerr) header(:), colorTable(:, :), rasterData(:)        ! BEWARE THAT PADDING IS READED !!!
    else
        read(10, rec = 1, err = 100, iostat = nerr) header(:), rasterData(:)                ! BEWARE THAT PADDING IS READED !!!
    end if
    close(10)

!    print *,rasterData(:)

    allocate(image(realImageSize))

!    do i = 1, height
!        do j = 1, line
!            image((i - 1) * line + j) = unsigned(rasterData((i - 1) * width + j))        ! do not include the padding !
!        end do
!    end do

    ! store the data into right order

    colbegin = 1 + ((height - 1) * width)
    do i = 0, line - 1
        do j = 0, height - 1
            image((i * height) + j + 1) = unsigned(rasterData(colbegin + i - (j * width)))    !do not include padding !!
        end do
    end do

!    do i = height, line * height, height
!        do j = 0, height - 1
!            image((i - height) + 1 + j) = rasterData(i - j)
!        end do
!    end do

    return

100     call errorFile()
200    call errorAlloc()

end subroutine extractRasterData

subroutine composeBMP(outputName)

! compose a BMP file with current header, color table and image (padding is considered already include in rasterData...)
! BESURE THAT RASTERDATA IS ALLOCATED AND ALL VARIABLES SET TO RIGTH VALUES! PLEASE, CALL THIS PROCEDURE AFTER A GOOD FILE ACQUISITION !

    implicit none
    character (len = 20) :: outputName
```

```
    integer :: i, j, colBegin

    open(20, file = outputName, status = 'unknown', err = 100, iostat = nerr, form = 'unformatted', &
            access = 'direct', recl = ths + cts + imageSize)

!    do i = 1, height
!        do j = 1, line
!            rasterData((i - 1) * width + j) = signed(image((i - 1) * line + j))
!        end do
!    end do

    colbegin = 1 + ((height - 1) * width)
    do i = 0, line - 1
        do j = 0, height - 1
            rasterData(colbegin + i - (j * width)) = signed(image((i * height) + j + 1))
        end do
    end do

! TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE

!    rasterData = image

! TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE TEMPORAIRE


    if (allocated(colorTable)) then
        write(20, rec = 1, err = 100) header(:), colorTable(:, :), rasterData(:)
    else
        write(20, rec = 1, err = 100) header(:), rasterData(:)
    endif

    close(20)

100    call errorFile()

end subroutine composeBMP

function f2o(i1, i2, i3, i4) ! convert four bytes in the corresponding number

    implicit none
    integer (kind = 4) :: f2o
    integer (kind = 2), intent(in) :: i1, i2, i3, i4

    f2o = i1 + (i2 * 256) + (i3 * 65536) + (i4 * 16777216)

    return

end function f2o

function checkColorTable()

    implicit none
    logical :: checkColorTable
    integer :: i, j
    integer (kind = 2) :: dummy
```

```
    do i = 1, 256
        dummy = i -1
        do j = 1, 3
            checkColorTable = checkColorTable .and. (colorTable(i, j) == signed(dummy))
        end do
    end do

    return

end function checkColorTable

function checkBMPType()

    implicit none
    logical :: checkBMPType

    checkBMPType = (header(compressionIndex) == compressionSupported) &
                .and. (header(bitCountIndex) == bitcountSupported)

    return

end function checkBMPType

function signed(x)

    integer (kind = 2), intent (in) :: x
    integer (kind = 1) :: signed

    if (x.gt.127) then
        signed = x - 256
    else
        signed = x
    end if

end function signed

function unsigned(x)

    integer (kind = 1), intent (in) :: x
    integer (kind = 2) :: unsigned

    if (x.lt.0) then
        unsigned = int(x, 2) + 256
    else
        unsigned = int(x, 2)
    end if

end function unsigned

subroutine errorFile()

    if (nerr .gt. 0) then
        print *,'file access error'
        stop
    else if (nerr .lt. 0) then
        print *,'end of file error'
```

```
        stop
    end if

end subroutine errorFile

subroutine errorAlloc

    if (ierr /= 0) then
        print *,'allocation error'
        stop
    end if

end subroutine errorAlloc


end Module BitmapHandler

module ImageCompressor

    use BitmapHandler
    use DWTransformer
    use Filtering

    implicit none

    public :: compress

CONTAINS

subroutine compress()

    ! execute DWT compression with all parameters stored in file param.dat
    ! store DWT in fileToCompress.dwt.bmp
    ! store DWT filtered in fileToCompress.dwt.filt.bmp
    ! store compress image reconstruction in fileToCompress.comp.bmp

    ! format of file :
    !     fileToCompress (without extention .bmp)
    !    motherWavelet
    !    ncoef
    !    filtering mode
    !    threshold


    character (len = 40) :: fileName, fileNameB
    character (len = 4) :: mw
    integer :: ncoeff, filtMode
    real :: thrh
    integer (kind = 2), allocatable, dimension(:) :: dataI
    real, allocatable, dimension(:) :: dataR
    integer, dimension(2) :: nIn
    integer, dimension(3) :: imSize

    open(50, file = 'param.dat')

    read(50, *)fileName
```

```
read(50, *)mw
mw = trim(mw)

read(50, *)ncoeff
read(50, *)filtMode
read(50, *)thrh

close(50)

fileNameB = fileName
fileNameB = trim(fileNameB)//'.bmp'

print *,fileNameB,'%%'

call extractRasterData(fileNameB)
imSize = getImageSize()
allocate(dataI(imSize(3)))
allocate(dataR(imSize(3)))
call getImageData(dataI)

nIn(1) = imSize(2)
nIn(2) = imSize(1)

dataR = dataI

print *,'Image ',fileNameB
print *,'computing DWT'
call DWT(dataR, nIn, mw, ncoeff, 1)
print *,'DWT done'
print *,'composing DWT image'
call quantif256(dataR, dataI)

call setImageData(dataI)
fileNameB = trim(fileName)//'.DWT.bmp'
call composeBMP(fileNameB)
print *,'file ',fileNameB,' composed'

print *,'filtering...'
call setFilteringParameters(dataR, thrh, filtMode)
call filter()
call getFilteringResult(dataR)

print *,'composing DWT filtered image'

call quantif256(dataR, dataI)

call setImageData(dataI)
fileNameB = trim(fileName)//'.DWT.filt.bmp'
call composeBMP(fileNameB)
print *,'file ',fileNameB,' composed'


print *,'computing DWT-1'
call DWT(dataR, nIn, mw, ncoeff, -1)
print *,'DWT-1 done'
```

```
    call quantif256(dataR, dataI)

    call setImageData(dataI)
    fileNameB = trim(fileName)//'.comp.bmp'
    call composeBMP(fileNameB)
    print *,'file ',fileNameB,' composed'

    return

end subroutine compress

end module ImageCompressor

program testA

use ImageCompressor

call compress()

!    character (len = 30) :: fileName, fileNameB
!    integer (kind = 2), allocatable, dimension(:) :: dataI
!    integer, dimension(3) :: imSize
!
!    open(50, file = 'param.dat')
!
!    read(50, *)fileName
!
!    close(50)
!
!    fileNameB = fileName
!    fileNameB = trim(fileNameB)//'.bmp'
!
!    print *,fileNameB,'%%'
!
!    call extractRasterData(fileNameB)
!    imSize = getImageSize()
!    allocate(dataI(imSize(3)))
!    call getImageData(dataI)
!
!    call setImageData(dataI)
!    fileNameB = trim(fileName)//'.B.bmp'
!    call composeBMP(fileNameB)
!    print *,'file ',fileNameB,' composed'

end program testA
```